Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2018

João Miguel Barraca Filipe

Redes Definidas por Software e Funções de Redes Virtualizadas para Comunicações Críticas e Fiáveis em Ambientes 5G

Software Defined Networks and Network Functions Virtualization for Critical and Reliable Communications in 5G Environments

Universidade de Aveiro
2018

Departamento de Eletrónica,
Telecomunicações e Informática

João Miguel Barraca Filipe

Redes Definidas por Software e Funções de Redes Virtualizadas para Comunicações Críticas e Fiáveis em Ambientes 5G

Software Defined Networks and Network Functions Virtualization for Critical and Reliable Communications in 5G Environments

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Rui Luís Andrade Aguiar, Professor Catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Daniel Nunes Corujo, Professor Investigador Doutorado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Professor Doutor Atílio Manuel da Silva Gameiro

Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Doutor Sérgio Miguel Calafate de Figueiredo

Consultor/Engenheiro Sénior, Altran Portugal

Prof. Doutor Rui Luís Andrade Aguiar

Professor Catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

agradecimentos /
acknowledgements

Agradeço ao Doutor Daniel Corujo e ao Professor Doutor Rui Aguiar por toda a orientação.
Agradeço a todos os meus colegas do IT por toda a ajuda, em especial ao Rui Silva, ao David Santos e ao Flávio Meneses.
Agradeço aos meus pais a paciência e o apoio financeiro.
Por último, um especial agradecimento aos meus amigos.
Dedico este trabalho à minha irmã e aos meus falecidos avós.

Palavras Chave

5G, SDN, NFV, Cloud, Virtualização, Container, VM, Unikernel.

Resumo

A Quinta Geração de Redes Móveis (5G), impulsionada pelo objetivo de conectar ainda mais a sociedade dos dias de hoje, terá que fazer uso de novas tecnologias emergentes como as Redes Definidas por Software (SDN) e Virtualização das Funções da Rede (NFV) para lidar com o enorme aumento de tráfego e serviços que estão a surgir. Os serviços Críticos e Fiáveis irão fazer uso destas tecnologias para criar novos mecanismos e/ou instanciar funções de rede que tenham requisitos muito rigorosos em ambientes virtualizados. O uso destas Funções de Rede Virtuais (VNFs) apresenta várias vantagens, como a rápida re-instanciação em caso de falhas ou a capacidade de serem escaladas nos serviços de Cloud fornecidos hoje em dia. Nesta dissertação é feito um estudo em que se compara o desempenho dos Containers e Máquinas Virtuais leves (Unikernels) para a instanciação de uma função de rede num ambiente virtualizado com recursos restritos. Também é implementado um mecanismo para garantir a fiabilidade da VNF. Os resultados demonstram que os Containers têm um melhor desempenho no caso de estudo apresentado e que os mecanismos de fiabilidade propostos asseguram a contínua actividade da VNF em caso de falha.

Keywords 5G, SDN, NFV, Cloud, Virtualization, Container, VM, Unikernel.

Abstract

The Fifth Generations of Mobile Networks (5G), driven by the aim to further connect today's society, will have to make use of new emerging technologies such as Software Defined Networking (SDN) and Network Functions Virtulization (NFV) to cope with all the increasing traffic and services that are arising. Critical and Reliable services will make use of these technologies to create new mechanisms and/or to instantiate network functions that have very strict requirements in virtualized environments. The use of these Virtual Network Functions (VNFs) presents several advantages like fast re-instantiation in case of failure or scaling capabilities that are provided by nowadays Cloud infrastructures. In this thesis a study is made comparing the performance of Containers and lightweight Virtual Machines (Unikernels) for the instantiation of a network function in a virtualized environment with restricted resources. It is also implemented a mechanism to ensure the reliability of the VNF. Results show that Containers perform better in the use-case presented and the proposed reliability mechanisms ensure zero downtime for the VNF in case of failure.

Contents

List of Figures

List of Tables

Glossary

2G Second Generation of Mobile Networks

3GPP 3rd Generation Partnership Project

4G Forth Generation of Mobile Networks

5G Fifth Generation of Mobile Networks

API Application Programming Interface

CAPEX Capital Expenses

CDN Content Delivery Network

CLI Command Line Interface

CPU Central Processing Unit

CSP Cloud Service Provider

DNS Domain Name System

DRAM Dynamic Random Access Memory

DSU Dynamic Software Updating

eMBB Enhanced Mobile Broadband

ETSI European Telecommunications Standards Institute

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

ICMP Internet Control Message Protocol

IDS Intrusion Detection System

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

ISP Internet Service Provider

KPI Key Performance Indicator

KVM Kernel-based Virtual Machine

LXC Linux Containers

M2M Machine-to-Machine

MAC address Media Access Control Address

mMTC Massive Machine Type Communications

MTC Machine-Type Communications

NaCl Not Another Configuration Language

NFV MANO NFV Management and Orchestration

NFVI Network Functions Virtualization Infrastructure

NFVO Network Function Virtualization Orchestrator

NFV Network Function Virtualization

NF Network Function

NIC Network Interface Controller

ONF Open Networking Foundation

OPEX Operating Expenses

OS Operating System

OvS Open vSwitch

PaaS Platform as a Service

PNF Physical Network Function

QoE Quality of Experience

RAM Random Access Memory

REST Representational State Transfer

RTT Round Trip Time

SaaS Software as a Service

SDN Software Defined Networking

SSD Solid State Drive

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UHD Ultra-High-Definition

URLLC Ultra-Reliable and Low Latency Communications

VIM Virtual Infrastructure Manager

VM Virtual Machine

VNFM Virtual Network Function Manager

VNF Virtual Network Function

1 | Introduction

1.1 Motivation

5G networks aim to enable a fully connected society and empower socio-economic transformations in several ways that are unpredictable. To achieve this all-connected society, the density/volume of traffic as well as connectivity is expected to grow tremendously [1]. This prediction of traffic increase is happening because 5G not only will support better human centric applications like virtual/augmented reality and Ultra-High-Definition (UHD) video but it also intends to make our life more convenient and comfortable by supporting the demands of machine-to-human and machine-to-machine type communications. With this large range of possible applications, 5G networks need to support a broad number of requirements [2].

To better achieve this variety of services and applications, new technologies are being introduced to edge and core networks, namely: SDN and NFV. With an architecture that makes use of these two technologies, the 5G network not only will provide better performance for the end user but it also provides a more flexible and dynamic provision of communications for different usage scenarios. Being one of these scenarios Critical and Reliable communications where SDN and NFV can really be differentiators, offering the ability to instantiate the necessary optimal network behavior at the right time.

Critical and Reliable communications were and still are a major concern to mobile operators. They are crucial for secure operation of Machine-Type Communications (MTC), such as monitoring and control systems, vehicle-to-vehicle coordination and cloud-based systems.

Some of these services have very strict requirements that lead to provision of communications in virtualized platforms between end devices and traditional computing data-centers. These platforms usually have limited resources and the instantiation of services needs to be done with lightweight virtualization techniques such as containers and unikernels.

This thesis addresses the role and impact that the enabling technologies for 5G referred above (SDN, NFV and other core/infrastructure enhancements) can have in enhancing and supporting services and applications that have reliable and critical data requirements in Machine-to-Machine (M2M) scenarios.

1.2 Objectives

This thesis aims to implement a Network Function (NF) in a data-center close to the end user with limited hardware resources. The implementation will be done using two virtualization

techniques, Hardware-Based and Operating System (OS) virtualization. The two techniques will be compared to see which one has a better performance regarding critical and reliable services.

A failure detection mechanism as well as a recover mechanism will also be implemented and tested to minimize the time that the NF is not operational after failing.

1.3 Document Structure

This document is structured as follows: Chapter 2 presents an overview on the state of the art and enabling technologies regarding cloud computing, SDN and NFV. Chapter 3 focuses on the description of the reliability environment and the proposed architecture design for the two implementations. In chapter 4, the performance results for both implementations are presented. Furthermore, an analysis and comparison of the results is made. Lastly, chapter 5 exhibits the final remarks: conclusion, contributions and future work.

2 | State of the Art and Enabling Technologies

2.1 Fifth Generation of Mobile Networks (5G)

In previous generations, from 2G to nowadays 4G, the main evolutionary purpose to new systems was focused towards increasing the number of users connected and providing higher data rates. 5G continues this trend, but it will be much more than "4G but faster" [3]. From the Figure 2.1 we can see that 5G has a lot to offer us in terms of services where three core targets for improving todays networks are clearly visible:

- Enhanced Mobile Broadband (eMBB) - or Extreme Mobile Broadband, this is the natural evolution of 4G networks. It aims to increase data rates and capacity providing a better Quality of Experience (QoE) to the users, broadband access to be available everywhere and higher user mobility with broadband services in moving vehicles like cars and public transportations.

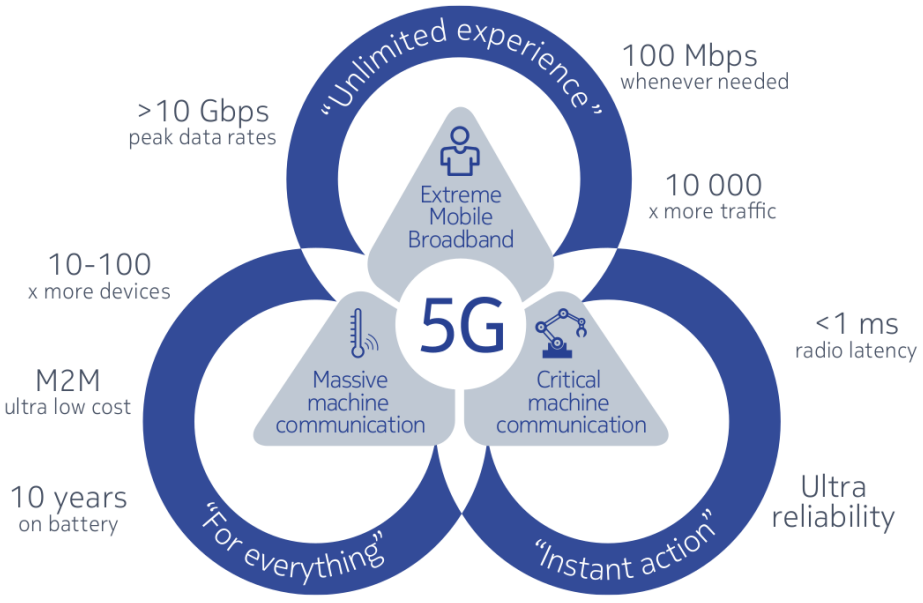

Figure 2.1: 5G Diversity of services, use cases and requirements [4]

- Ultra-Reliable and Low Latency Communications (URLLC) - this is the most innovative feature in 5G. It will be used for mission critical services and ultra reliable communications that require very strict Key Performance Indicators (KPIs). This feature can enable new types of services like the Industry 4.0 with automation, vehicles coordination and real-time remote control [5]. Since URLLC is a very embryonic type of communications, 3rd Generation Partnership Project (3GPP) came up with six different types of use cases given that not all services need the same KPIs [6]:
  - Higher reliability and lower latency
  - Higher reliability, higher availability and lower latency
  - Very low latency
  - Higher accuracy positioning
  - Higher availability
  - Mission Critical Services

  The latency requirements for the use cases above vary from under 1 ms to 10 ms and the reliability specifications can range from five nines (99.999%) to nine nines (99.9999999%).

- Massive Machine Type Communications (mMTC) - this type of service was created to support the access of a large number of MTC devices and services like monitoring, sensing, tagging and metering. All the previous services require high connection density but don't require very strict latencies such as URLLC or high bit-rates as eMBB [7].

To cope with this wide range of services and applications, 5G networks need to turn to new emerging technologies like SDN, NFV and Cloud/Edge Computing. Using these new enabling technologies, mobile networks can increase their flexibility, scalability while reducing the overall cost of the network [8].

2.2 Software Defined Networking (SDN)

A new emerging network architecture for mobile networks is SDN, where the network control plane is decoupled from the data plane (also known as forwarding plane). With the decoupling of these two planes, the behavior of the network equipments is controlled by a logically centralized controller that has a full view of the network, providing high level of network programmability and allowing a dynamic network (re)configuration. Figure 2.2 summarizes the SDN architecture abstractions in form of a detailed, high-level schematic. Starting from the top of the figure and moving towards the lower part, the following planes are identified [9]:

- Application Plane - Services and applications that define network behavior are located in this plane;
- Control Plane - This plane is responsible for making and pushing decisions on how packets should be forward by one or more network devices. Although the control plane may be interested in operation-plane information like the current state of a particular

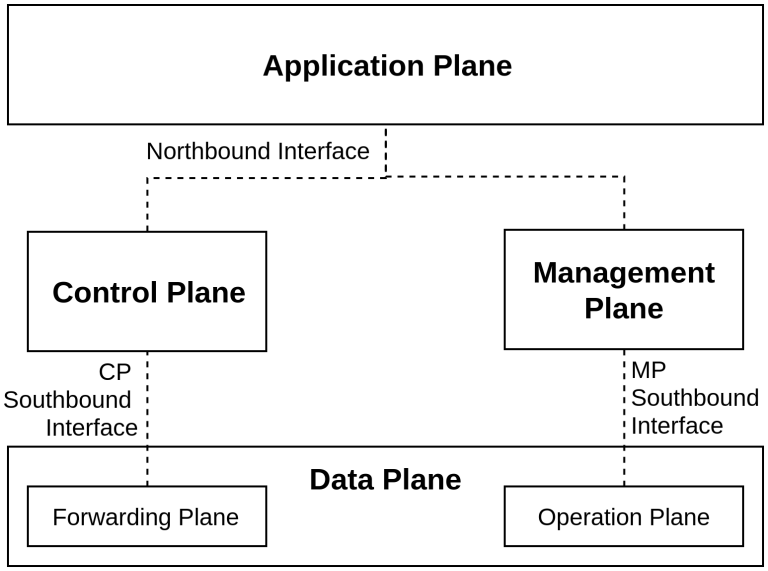

Figure 2.2: SDN Layer Architecture

port, it usually focuses mostly on the forwarding plane. The main job of the control plane is to fine-tune the forwarding tables presented in the forwarding plane, based on the network topology and/or external service and application requests. The Control Plane receives information from services in the Application Plane through the Northbound Interface. Examples of protocols used for the Northbound interface are RESTful APIs;

- Management Plane - Contrary to the control plane, the management plane mainly focuses on the operation plane. It is responsible for monitoring, configuring, and maintain network devices;
- Operation Plane - The operational plane is usually the termination point for management services and applications. It serves the purpose of managing the operational state of the network devices like the number of ports available, the status of each port and the status of the device - active or inactive. It receives information from the Management Plane Southbound Interface and the protocols used in this interface are vendor specific;
- Forwarding Plane - This plane is usually the termination point for control-plane services and applications. It is responsible for handling packets in the data plane based on the instructions received from the control plane. Actions in this plane include forwarding, dropping and changing packet's headers. These actions are performed based on the rules provisioned by the Control Plane Southbound Interface. ForCES [10], YANG model [11] and OpenFlow are examples of protocols used for this interface, being OpenFlow the most used.

2.2.1 OpenFlow Protocol

OpenFlow is a open-source protocol that enables network controllers to handle the network traffic by establishing communication between the control application layer and the forwarding plane, handling the network as a whole rather than individual devices, promoting a more

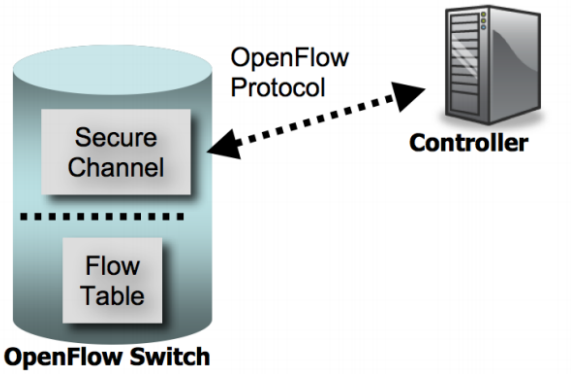

Figure 2.3: OpenFlow Architecture

efficient use of network resources. It was one of the first protocols specially designed for SDN and is standardized by the Open Networking Foundation (ONF) [12].

The baseline architecture for OpenFlow is shown in Figure 2.3 and consists in three key elements - an OpenFlow controler, an OpenFlow switch and a secure communication channel for the controller-switch communication [13].

2.2.2 OpenFlow Switches

An OpenFlow Switch, presented in Figure 2.4, consists of one or more flow tables and a group table which perform packet forwarding and lookups, and a OpenFlow Channel that connects to an external controller. An OpenFlow switch may establish communication with one or more controllers. The communication between the switch and the controller happens via the OpenFlow Protocol [14]. This communication is established using a specific IP and port, initializing a Transmission Control Protocol (TCP) or Transport Layer Security (TLS) session between the swtich and the controller. The traffic between the switch and the controller does not go through the OpenFlow pipeline. At the time of this writing there are several OpenFlow switches, being Open vSwitch (OvS)[1] one of the most popular ones.

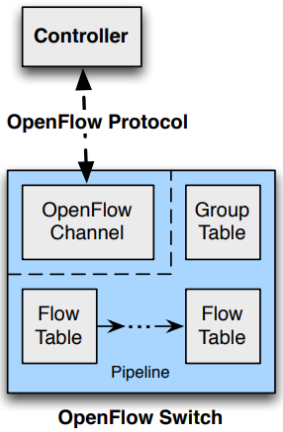

Figure 2.4: OpenFlow Switch

[1] http://www.openvswitch.org/

2.2.3 OpenFlow Controllers

The OpenFlow controller, like a regular SDN controller, has a global logical view of the network. It is responsible for handling the flow tables of the OpenFlow switch and handle traffic without valid flow entries.

Up to the time of this writing, there are several open-source controllers such as: OpenDaylight [2], Floodlight [3], ONOS [4], POX [5] and Ryu [6].

2.3 Virtualization

Virtualization is the process of running a virtual instance of a computer system in a layer abstracted from the actual hardware. It allows you to use a physical machine's full capacity by running multiple OSs instances sharing the underlying hardware [15]. Virtualization can be divided in different types:

- Data virtualization - with data virtualization companies can treat data as a dynamic supply, bringing together data from multiple sources, integrate new data and transform data according to user needs;
- Network Function Virtualization (NFV) - NFV separates networks functions like firewalling, Intrusion Detection System (IDS) and Domain Name System (DNS), from proprietary/specific hardware so they can run as software. NFV is detailed in section 2.3.5;
- OS virtualization - also known as containerization, this type of virtualization happens at the central task manager of the system - the kernel. The kernel allows the co-existence of multiple isolated user-space instances. Usually these instances are called containers. Containers are presented with more detail in section 2.3.3;
- Desktop virtualization - this type of virtualization is usually confused with OS virtualization. In desktop virtualization, a central administrator can deploy simulated desktop environments to hundreds of physical machines at once. This ability allows the administrator to launch updates, configurations and security checks on all the virtual desktops;
- Server virtualization - This virtualization aims to increase resource sharing and allows, in theory, to create enough virtual servers to use all of a physical machine's processing power, thus optimizing the total hardware of a physical server.

In 1974, Gerald J. Popek and Robert P. Goldberg defined a VM as an efficient, isolated duplicate of a real computer machine [16]. A VM, also known as a guest, is a software program or OS that supports individual processes or a complete system. They are based on computer architectures and provide the functionality of a physical computer. VMs can be classified in

[2] https://www.opendaylight.org/
[3] http://www.projectfloodlight.org/floodlight/
[4] https://onosproject.org/
[5] https://github.com/noxrepo/pox
[6] https://osrg.github.io/ryu/

two types. The first one being System VMs, also known as full virtualization VMs, mirrors all the components and processes of a physical machine, providing the needed functionalities of a full OS. The second type are Process VMs, these VMs are designed to execute computer programs in a platform-independent environment.

Hypervisor is a program that allows the creation and instantiation of VMs. They separate the physical resources such as Central Processing Units (CPUs) memory, I/O and network traffic to be used by these virtual instances. There are two classes of hypervisors [17]:

- Bare metal or type 1 hypervisors, behave as an OS and run guest VMs directly on the system's hardware;
- Hosted or type 2 hypervisors, behave as a normal program and can be started and stopped like any traditional application.

Nowadays the difference between these two types of hypervisors is very thin, especially with technologies like KVM.

2.3.1 KVM

Kernel-based Virtual Machine (KVM) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). Using KVM, one can run multiple VMs on the same physical machine. Each virtual machine has private virtualized hardware: a network card, disk, graphics adapter, etc. [18].

KVM converts Linux into a type-1 hypervisor. Considering that KVM is part of the Linux kernel (since version 2.6.20) it has all the key components of a OS. Every VM instantiated is implemented as a regular Linux process, scheduled by the standard Linux scheduler, with dedicated virtual hardware like a network card, graphics adapter (if needed), CPUs, memory, and disks [19].

2.3.2 Unikernels

Unikernels are specialized, single-address-space machine images constructed by using library operating systems [20]. They are specialized because unikernels were designed to hold and run a single application/service that is compiled into standalone kernels, including only the functions and routines needed to perform the appliance, and sealed against modification when deployed to a cloud platform. In return, unikernel applications offer significant reduction in image sizes, improved efficiency and security, and should reduce operational costs. The entire software stack of system libraries, language runtime, and applications is compiled into a single bootable VM image that runs directly on a standard hypervisor [21].

In contrast to standard VMs and containers, unikernels present an advantage in terms of security. Since unikernel services are compiled with the minimal components to run a specific task, the attack surface is reduced in comparison to the other two virtualization platforms, where a lot of unnecessary components are instantiated, increasing the attack surface.

At the time of this writing there are several unikernel projects, such as:

- *ClickOS* - A high-performance, virtualized software middlebox platform. ClickOS virtual machines are small, boot quickly, with little delay and over one hundred of them can be concurrently run while saturating a 10Gb pipe on a commodity server [22];

- *Clive* - Clive is an operating system designed to work in distributed and cloud computing environments [23];
- *Drawbridge* - Drawbridge is a research prototype of a new form of virtualization for application sandboxing. Drawbridge combines two core technologies: First, a picoprocess, which is a process-based isolation container with a minimal kernel API surface. Second, a library OS, which is a version of Windows enlightened to run efficiently within a picoprocess [24];
- *HaLVM* - The Haskell Lightweight Virtual Machine (HaLVM) is a port of the Glasgow Haskell Compiler toolsuite that enables developers to write high-level, lightweight virtual machines that can run directly on the Xen hypervisor [25];
- *IncludeOS* - IncludeOS is a minimal, open source, unikernel operating system for cloud services [26]. This project is detailed in subsection 2.3.2.1;
- *LING* - A unikernel based on the Erlang/OTP and understands .beam files. Developers can create code in Erlang and deploy it as LING unikernels. LING removes the majority of vector files, uses only three external libraries and no OpenSSL [27];
- *MirageOS* - MirageOS is a library operating system that constructs unikernels for secure, high-performance network applications across a variety of cloud computing and mobile platforms. Code can be developed on a normal OS and then compiled into a fully-standalone, specialised unikernel that runs under a Xen or KVM hypervisor [28];
- *OSv* - OSv is designed from the ground up to execute a single application on top of any hypervisor, resulting in superior performance, speed and effortless management. Can run unmodified Linux executables (with some limitations), and support for C, C++, JVM, Ruby and Node.js application stacks is available [29];
- *Rumprun* - A software stack which enables running existing unmodified POSIX software as a unikernel. Rumprun supports multiple platforms, including bare hardware and hypervisors such as Xen and KVM [30];
- *runtime.js* - runtime.js is an open-source library operating system (unikernel) for the cloud that runs JavaScript, can be bundled up with an application and deployed as a lightweight and immutable VM image [31];

As for the unikernel chosen for this thesis, two projects were initial selected - IncludeOS and MirageOS. They were picked for their versatility, as well as documentation available. Although the MirageOS project provided a lot of documentation, the implementation of this technology was not successful due to compatibility issues with the OS. IncludeOS proved to be very simple to install and although a few setbacks were encountered, the help from the developers and maintainers of the open-source project were crucial for defining IncludeOS as the unikernel chosen for this work.

2.3.2.1 IncludeOS

IncludeOS is a single-tasking OS created for virtualized environments. With this unikernel technology, developers can, at compile-time, build their C++ based applications directly into a VM. It presents three key elements to nowadays cloud platforms [32]:

- *Resource Efficiency and Footprint* - Similarly to other unikernel technologies, with IncludeOS only the necessary parts for the designed service are included, reducing waste and improving network and memory performance;
- *Efficient Deployment Process* - It uses a custom GCC-based[7] toolchain. During link time, a single binary is formed using the required elements from the pre-compiled OS-library. A boot sector is also attached, resulting in a bootable disk image;
- *Virtualization Platform Independence* - IncludeOS unikernels are written to run on virtualized x86 hardware. The resulting disk image from the deployment process can be uploaded to OpenStack[8] or automatically formatted to fit most virtualization environments like KVM.

From the literature [32], IncludeOS was compared to traditional Linux VMs and present several advantages like:

- Lightweight disk, low memory footprint and overall performance increase, due to simplistic design;
- Since the OS and the service are the same binary, there is no system call overhead, as the system calls are simple function calls;
- Reduced number of VM exists by keeping the number of protected instructions very low.

2.3.3 Containers

As it was previously mentioned, containers are a method of OS virtualization. Containers are a logical package mechanism in which applications and services can be abstracted from the underlying infrastructure. This abstraction allows container-based application to be deployed easily through a variety of environments, such as private data centers, public clouds or even personal computers [33].

There are several benefits for using containers [34]:

- Environment Consistency - Since containers encapsulate all the necessary service files, libraries and software dependencies into a block, deployment becomes platform independent as these blocks can be deployed regardless of the OS or hardware configurations. This allows service providers to release new features and updates faster as testing a container locally will behave and run in the same way as in production;
- Operational Efficiency - By allowing multiple containers on the same host, containerization increase the efficiency of computing resources. It is even possible to specify the exact amount of memory, disk space, and CPU to be used by a container. Since each container is only a process on the OS running an application and its dependencies they have very reduced footprints and therefore are able to achieve very fast boots. They also provide process isolation;
- Developer Productivity - Containers increase developer productivity by removing cross-service dependencies and conflicts. Every service component can be broken into

[7] http://gcc.gnu.org/
[8] https://www.openstack.org/

different containers running a different microservice. Since containers are isolated from one another, libraries or dependencies being in sync for each service is not a problem. Developers can independently upgrade each service because there are no library conflicts.

At the time of this writing, there are several container solutions available. A few examples are:

- *Linux Containers (LXC)* - LXC is a userspace interface for the Linux kernel containment features. Through a powerful API and simple tools, it lets Linux users easily create and manage system or application containers [35];
- *Warden Container* - Warden container provides a kernel independent containment implementation which can be plugged to multiple underlying Host OS [36]. This container implementation is used by Cloud Foundry project to host applications [37];
- *Docker* - Detailed in subsection 2.3.3.1;
- *OpenVZ* - OpenVZ uses a modified Linux Kernel with a set of extensions. OpenVZ manages multiple physical and Virtual servers, by using dynamic real-time partitioning [36]. OpenVZ is the core of a virtualization solution offered by the Virtuozzo company [38].

2.3.3.1 Docker

Docker is the most popular open-source approach for application oriented containers. It makes use of Linux kernel features, such as namespaces and control groups, to isolate independent containers running on the same instance of the operating system [39]. All the necessary parts for running the container, such as code, runtime, system tools, system libraries and settings, are packed in a lightweight software package called Docker container image. This container image becomes a container when they run on Docker Engine.

The Docker Engine is available for both Linux and Windows-based applications and containerized software will always run the same, regardless of the underlying infrastructure [40].

Docker provides two main services: Docker Engine Enterprise and Docker Engine Community, being the first a commercially-supported enterprise engine and the second a free community-supported engine.

2.3.4 Virtualization Comparison

The three virtualization solutions presented, VMs, containers and unikernels, have several differences. Table 2.1 presents a summary of the pros and cons of these architectures. Figure 2.5 displays a visual comparison of the three.

	Pros	Cons
VMs	- Isolation from host; - Orchestration solutions available.	- High memory overhead; - Number of instances can be a problem; - A whole OS is virtualized.
Containers	- Fast boots; - Orchestrations solutions available; - Dynamic resource allocation; - Lightweight virtualization.	- Low isolation between the containers and the host as the kernel is shared; - Less secure.
Unikernels	- Lightweight VM; - Specialized machine; - Isolation from host; - Higher security.	- Limited deployment possibilities; - Resource allocation is static; - Lack of orchestration solutions; - Low maturity for production.

Table 2.1: Pros and Cons of VMs, containers and unikernels

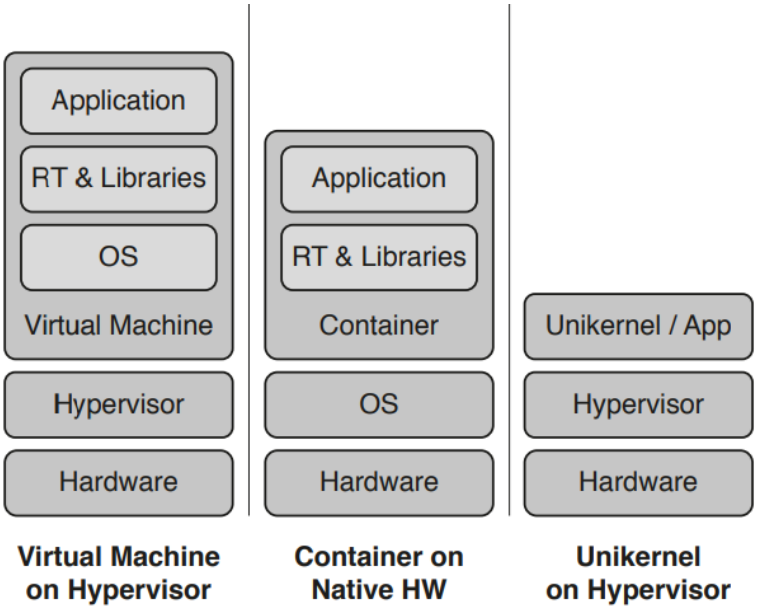

Figure 2.5: Illustrated comparison of VMs, containers and unikernels [39]

2.3.5 Network Function Virtualization (NFV)

NFV aims to transform the way network operators architect nowadays networks by evolving standard virtualization technologies to consolidate specialized network equipment, such as firewalls, Content Delivery Networks (CDNs), IDSs, etc., onto industry standard high volume server, switches and storage as VNFs. This centralization can happen at Data centers, network nodes or at the end user premises.

By decoupling NFs from the physical devices on which they run, NFV has the potential to lead to significant reductions in Operating Expenses (OPEX) and Capital Expenses (CAPEX) and facilitate the deployment of new services with increased agility and faster time-to-value [41].

NFV introduces a number of differences in the way network services are provisioned in comparison to current practice. European Telecommunications Standards Institute (ETSI)

listed three main differences [42]:

- *Decoupling software from hardware* - As the network element is no longer a collection of integrated hardware and software entities, the evolution of both is independent of each other. This enables the software to progress separately from the hardware, and vice versa;
- *Flexible network function deployment* - The detachment of software from hardware helps reassign and share the infrastructure resources, thus together, hardware and software, can perform different functions at various times. The actual network function software instantiation can become more automated, leveraging the different cloud and network technologies currently available. This helps network operators deploy new network services faster over the same physical platform;
- *Dynamic operation* - The decoupling of the functionality of the network function into software components that can be instantiated, provides greater flexibility to scale the actual VNF performance in a more dynamic way and with finer granularity, for instance, according to the actual traffic for which the network operator needs to provision capacity.

2.3.5.1 NFV Architecture

ETSI defined the NFV architectural framework with three main elements, the NFV Management and Orchestration (NFV MANO), Network Functions Virtualization Infrastructure (NFVI) and the VNFs and services, as shown in Figure 2.6.

2.3.5.1.1 VNFs and Services.

As it was previously mentioned, a VNF is a functional block within a network infrastructure that was deployed in virtual resources. The functional behavior, external operational interfaces and state of the NF are expected to be the same for Physical Network Function (PNF) and for a VNF.

VNFs can be composed of multiple internal components. For example, one VNF can be deployed over multiple VMs, where each VM hosts a single component of the VNF. However, in other cases, the whole VNF can be deployed in a single VM as well [42]. As an alternative the virtualization platform can be switched to containers or even unikernels.

A service usually is provided by a network operator and can be decomposed into a set of VNFs. Nonetheless, in the users' perspective, the services should have the same or better performance, whether running in traditional PNFs or in VNFs [43].

2.3.5.1.2 Network Functions Virtualization Infrastructure (NFVI).

For the instantiation of VNF a pool of both hardware and software resources is needed. The NFVI is the combination of these resources. The physical resources consist of both computing hardware, storage and network, that provide storage, connectivity to the VNFs. Software resources usually consist of hypervisors that abstract the underlying hardware and/or software resources they run on.

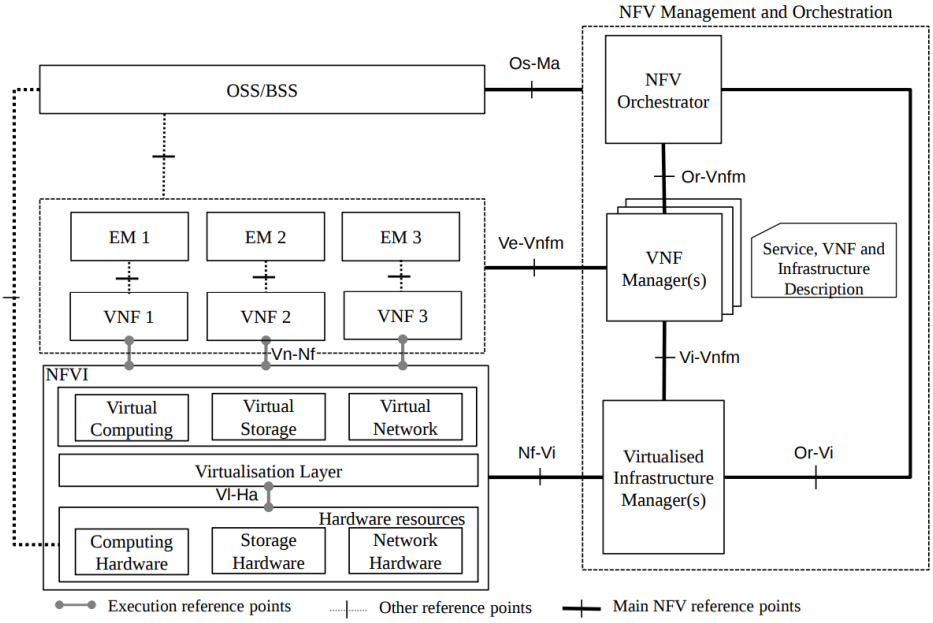

Figure 2.6: NFV reference architectural framework [42]

2.3.5.1.3 NFV Management and Orchestration (NFV MANO).
To enable network automation, the NFV MANO was introduced. It provides the functionalities required for the provisioning, managing and configuration of the VNFs. The NFV MANO is composed by three blocks: the Network Function Virtualization Orchestrator (NFVO), the Virtual Network Function Manager (VNFM) and by the Virtual Infrastructure Manager (VIM) [42].

NFVO. The NFVO is in charge of the orchestration and lifecycle management of the NFVI and physical and/or software resources.

VNFM. Oversees lifecycle management of VNF instances, such as instantiation, update, query, scaling, termination, etc.. Multiple VNFM may be deployed as a VNFM may be deployed for each VNF, or may serve multiple VNFs.

VIM. The VIM controls and manages the interaction between the VNFs and the computing, network and storage resources. It is responsible for resource management like inventory of software, allocation of VMs and allocation of resources. In terms of operation it manages failure information, capacity planning and optimization.

2.4 Cloud Computing

Cloud computing refers to both the applications delivered as a service over the Internet and the hardware and software systems in the data centers that provide those services [44]. The National Institute of Standards and Technology (NIST) defined cloud computing as follows: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access

to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction."
This model aims to make a better use of various computer resources, achieve higher throughputs and to resolve problems that require high performance computations. It is composed by five characteristics, four deployment models and three service models [45]:

- Characteristics:
  - On-demand self-service - Computing capabilities such as network storage and server time can be provision by a consumer, without any human interaction;
  - Broad network access - Cloud resources are available over the Internet and accessed through different devices (e.g., mobile phones, tablets, computers, etc.);
  - Resource pooling - A multi-tenant model is used to serve multiple consumers with the provider's computing resources. The consumer has the possibility to choose between physical and virtual resources which are assigned and/or reassigned dynamically. Usually the exact location of the provided resources is uncontrolled by the customer, however the user may be able to specify the location at a higher level of abstraction (e.g., country, data center, etc.);
  - Rapid elasticity - Provisioned resources can be elastically created, scaled and released to meet consumer demands;
  - Measured service - Resource usage can be controlled, monitored and reported providing transparency for both the consumer and the provider of the service.

- Deployment Models:
  - Private cloud - Usually this cloud infrastructure is provisioned for an exclusive client or single organization with multiple consumers;
  - Community cloud - This type of cloud infrastructure is provisioned for a specific community of consumers;
  - Public cloud - Cloud infrastructure provision to the general public.
  - Hybrid cloud - This type of infrastructure is a combination of two or more distinct cloud infrastructures referred above.

- Service Models:
  - Software as a Service (SaaS) - The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure;
  - Platform as a Service (PaaS) - The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider;
  - Infrastructure as a Service (IaaS) - The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications.

2.5 Summary

This chapter presented a brief overview of enabling technologies for 5G networks, namely SDN and NFV and cloud computing. It also presented the state of the art and available open-source projects for different virtualization technologies such as containers and unikernels.

In the following chapter a use-case scenario is presented as well as the overall architecture design for the evaluation of this work.

3 | Scenario Description and Proposed Architecture

This chapter presents a description of the use-case scenario and the architecture design for testing the overall performance of the VNFs implemented using two different virtualization technologies.

3.1 Scenario

For the critical/reliability use-case scenario, the perspective of an Internet Service Provider (ISP)/Cloud Service Provider (CSP) that is hired by a hypothetical industrial corporation to host critical VNFs in their behalf is considered. Due to the nature of these VNFs, they need to be managed as black boxes, which means that the CSP cannot modify or alter in any way the operation of the VNFs. The service provider can only interact with the VNFs via a well specified API. Through this API, the industrial corporation needs to be able to send new versions of the VNFs to the CSP so that the older versions are updated. This procedure is illustrated in Figure 3.1.

The CSP is also responsible for ensuring the reliability of the VNFs as well as providing near zero-time between version updates and failures.

In this particular use-case, a critical network firewall is provisioned. Due to the nature of the system where the firewall is placed, there is the need for a more flexible way to allow new versions of the filter rules to be deployed in the firewall. It is important to consider that the

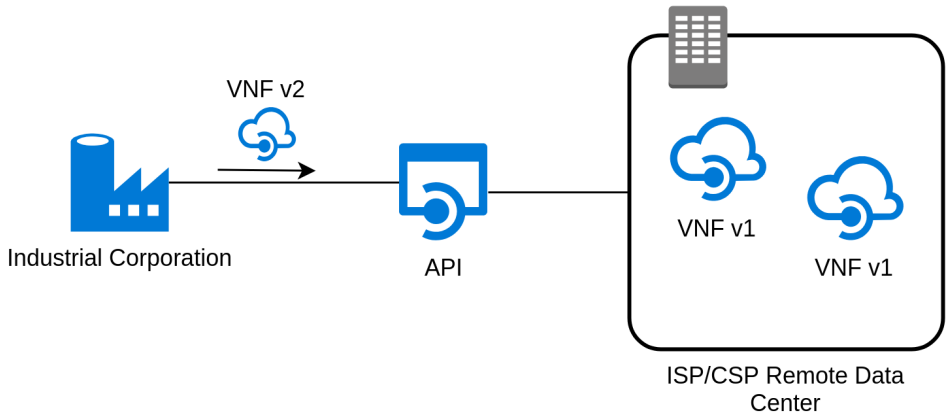

Figure 3.1: Illustrated use-case scenario

black-box nature requirement of the VNF prevents the injection of new filter rules directly in the firewall. As such, a new version of the firewall - with the new rules - needs to be deployed.

To be able to cope with the reliability and latency requirements imposed by the industrial corporation, the cloud provider needs to instantiate the VNFs in a remote data-center close to the company. Since it is not a big data-center, there are hardware resource limitations that increase the probability of failure [46]. To handle with this possibility of failures, the CSP needs to develop a failure detection mechanism that reduces the time the firewall is down when a disruption occurs.

3.2 Architecture

Figure 3.2 presents the high-level design used to simulate and validate the use-case scenario in 3.1.

3.2.1 Remote Data Center

In this architecture the remote data center is represented by an apu2c4[1], a single board computer developed by PC Engines[2]. This device has an AMD GX-412TC SOC CPU with 4 cores, 4 GB DDR3-1333 DRAM and a 16 GB SSD for storage. The OS installed was the Ubuntu 17.10 with the 4.13.0-21-generic kernel. After the OS installation, the system was turned into a type-1 hypervisor with KVM, allowing the host machine to run multiple isolated virtual environments that we call VMs or guests.

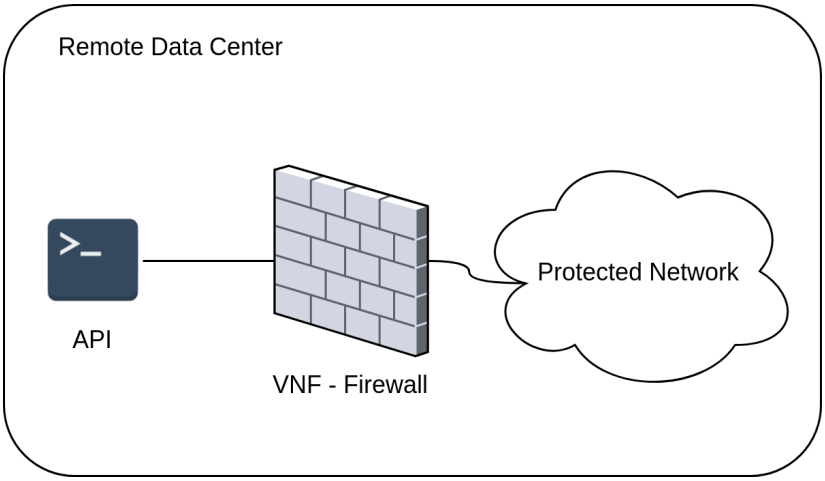

Figure 3.2: High-level design

3.2.2 API

To simulate the API, the Command Line Interface (CLI) of the data-center is used. In a commercial situation, this API could be a Representational State Transfer (REST) service

[1] https://pcengines.ch/apu2c4.htm
[2] https://www.pcengines.ch/index.htm

but its implementation is not in the scope of this thesis.
As it can be seen in Figures 3.3 and 3.6, the bridge br-dabc81852bad was created inside the data center. This bridge is needed to connect the IncludeOS Firewall to the host network. Since it is needed by the unikernel, when the VNF is instantiated in a container, the Docker container will also be attached to this bridge to match the two architectures. It represents the connection point between the external public network and the VNF. It has the 02:42:9b:2f:a1:76 MAC address and was assigned the 10.0.0.1 Internet Protocol (IP) address.

3.2.3 Protected Network

Similarly to the API, the protected network is represented by a linux bridge - br-8c1fd3936e33 - which has the 02:42:80:2d:42:45 MAC address and the 192.168.0.1 IP address assigned. This bridge is needed by the same purpose as in 3.2.2, is completely isolated from the outside network since all routes from the host were removed and it only has two connection points: the VNF and a sink node.

The sink node is a Docker container created with a Dockerfile shown in appendix A, based on the ubuntu-nettools image created by Robertxie[3]. This sink node is also isolated from the outside network as it is only connected to the br-8c1fd3936e33 bridge. It does not fit any purpose other than receiving packets to test the VNF.

3.2.4 Database

The purpose of this database is to store different versions of the VNF. When the VNF is updated, the database updates the current-version file with the new version. It runs as a docker container, with an image similar to the one used by the sink node, that runs a simple python HTTP server[4]. The database is only accessed when the VNF is running in its stateful configuration. As it can be seen with dashed lines in Figure 3.3 and 3.6, the database is connected to the external network. Configuration files are shown in appendix B.

3.2.5 VNF - Firewall

This is the critical VNF that will be tested in several different metrics to determine which virtualization platform has an overall better performance. The virtual firewall will be instantiated in a Docker container and in a IncludeOS unikernel, and will be more detailed in the following subsections:

3.2.5.1 Docker Firewall

The Docker Firewall is an Alpine container[5] with the iptables[6] package. With these modifications (presented in the Dockerfile in appendix C), the image size of the firewall is 16.4 MB. It is based on iptables rules that filter the packets according to the configuration. In its stateless configuration, the firewall always has the same version when it is launched (version 0). Regarding the stateful configuration, when the docker firewall is instantiated it downloads

[3]https://hub.docker.com/r/robertxie/ubuntu-nettools/
[4]https://docs.python.org/3/library/http.server.html
[5]https://hub.docker.com/_ /alpine/
[6]https://netfilter.org/projects/iptables/

the current-state file from the database, launching the container with the last updated version. The firewall has two Network Interface Controllers (NICs):

- eth0 with the 10.0.0.2 IP address, connected to the public network (br-dabc81852bad bridge);
- eth1 with the 192.168.0.3 IP address, connected to the private network (br-8c1fd3936e33 bridge).

This Docker container is launched using the docker-compose[7] tool, allowing some firewall configurations such as resources limits and network configurations to be predefined in a docker-compose.yml file (full file also in appendix C). A full architecture is represented in Figure 3.3.

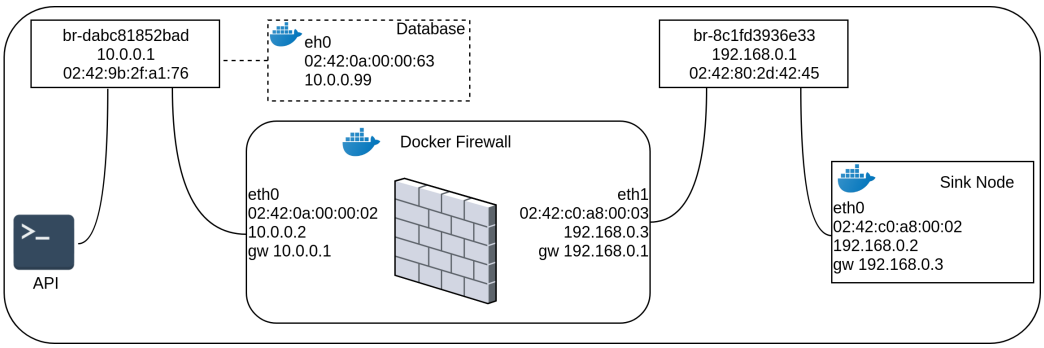

Figure 3.3: Detailed design with Docker Firewall

3.2.5.1.1 Instantiation Procedure.

A) Stateless.

Figure 3.4 illustrates the instantiation procedure of the Docker Firewall in its stateless configuration.

This procedure is described as follows:

1. The API sends the instantiation command to the Docker daemon inside the data-center;
2. Upon receiving the command, the daemon instantiates the Docker Firewall in the host (data-center).

The Stateless Docker Firewall is now ready.

[7] https://docs.docker.com/compose/overview/

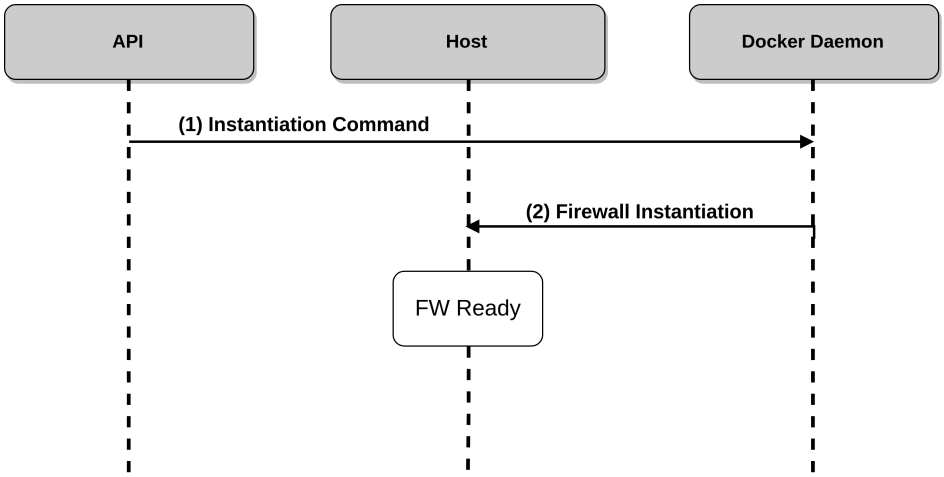

Figure 3.4: Stateless Docker Firewall Instantiation Procedure

B) Stateful.

The stateful instantiation procedure is presented in Figure 3.5 and is described as follows:

1. The API sends the instantiation command to the Docker daemon inside the data-center;
2. After receiving the command, the daemon instantiates the stateful Docker Firewall in the host;
3. The firewall is instantiated and requests the current-version file from the Database;
4. The Database sends the file to the firewall;
5. Finally, upon receiving the file, the Docker Firewall updates its policies.

The VNF is now ready.

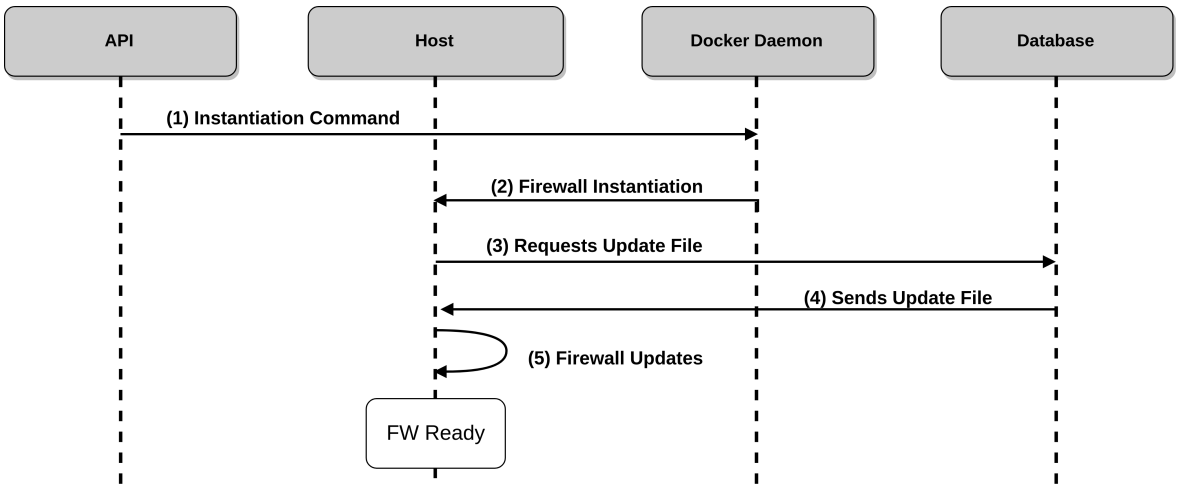

Figure 3.5: Stateful Docker Firewall Instantiation Procedure

3.2.5.2 IncludeOS Firewall

Contrary to the Docker Firewall, the IncludeOS Firewall is a unikernel. It is compiled into a binary file via a C++ code where the firewall rules are programmed. To facilitate the writing of the C++ program, IncludeOS provides its own configuration language - Not Another Configuration Language (NaCl)[8]. The full configuration file of the firewall using the NaCl language is presented in appendix D. When the binary file is generated after the compilation, the VM (unikernel) is launched by QEMU[9] taking advantage of the KVM acceleration provided by the host machine (remote data center - 3.2.1).

The IncludeOS Firewall also shares the same database as the configuration in 3.2.5.1. When running in its stateful mode, the hypervisor downloads the latest binary updated to the database container, before launching the unikernel.

Similarly to the Docker Firewall, the IncludeOS Firewall also has two NICs:

- eth0 connected to the public network (br-dabc81852bad bridge);
- eth1 connected to the private network (br-8c1fd3936e33 bridge).

The IP addresses, MAC addresses as well as the full architecture is presented in Figure 3.6:

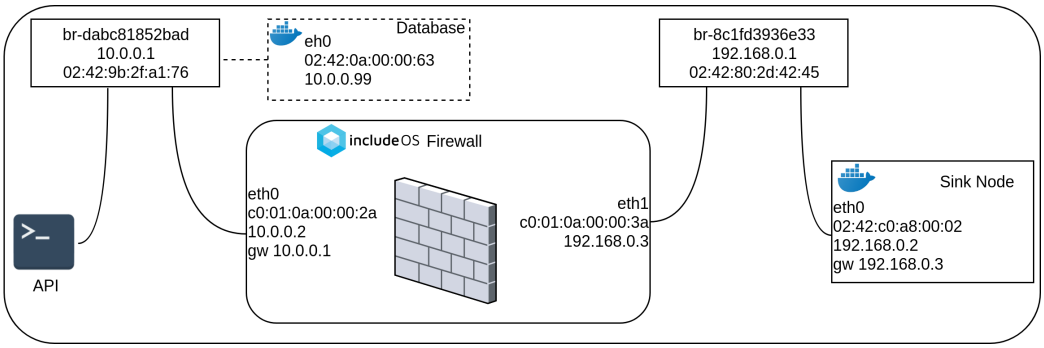

Figure 3.6: Detailed design with IncludeOS Firewall

3.2.5.2.1 Instantiation Procedure.

A) Stateless.

The intantiation of the IncludeOS Stateless Firewall is described in Figure 3.7.

1. The API sends the command to the hypervisor to instantiate the firewall;
2. The host launches the firewall with QEMU.

After QEMU successfully launches the unikernel, the VNF is ready.

[8]https://includeos.readthedocs.io/en/latest/NaCl.html
[9]https://www.qemu.org/

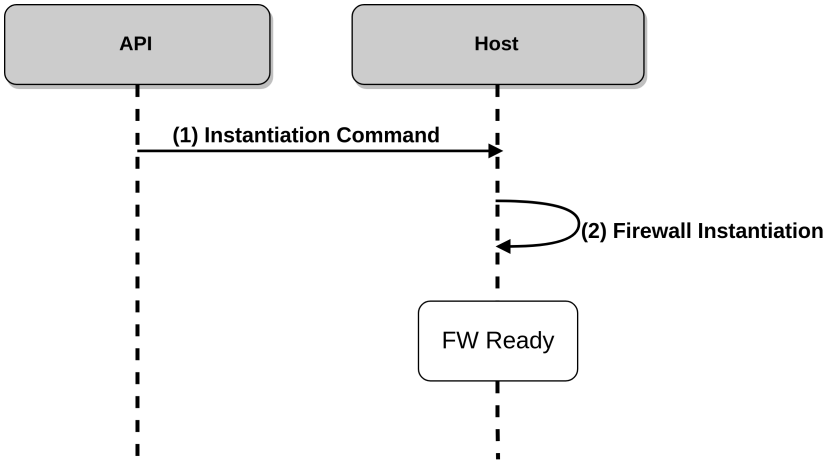

Figure 3.7: Stateless IncludeOS Firewall Instantiation Procedure

B) Stateful.

Figure 3.8 presents the stateful instantiation procedure for the VNF.

1. Similarly to the stateless procedure, the Application Programming Interface (API) sends the command to the host;
2. Before launching the unikernel, the hypervisor requests the latest updated version to the Database;
3. The Database sends the latest firewall version to the host;
4. Finally, upon receiving the file, the unikernel is instantiated with the last updated version.

The IncludeOS Firewall is now ready.

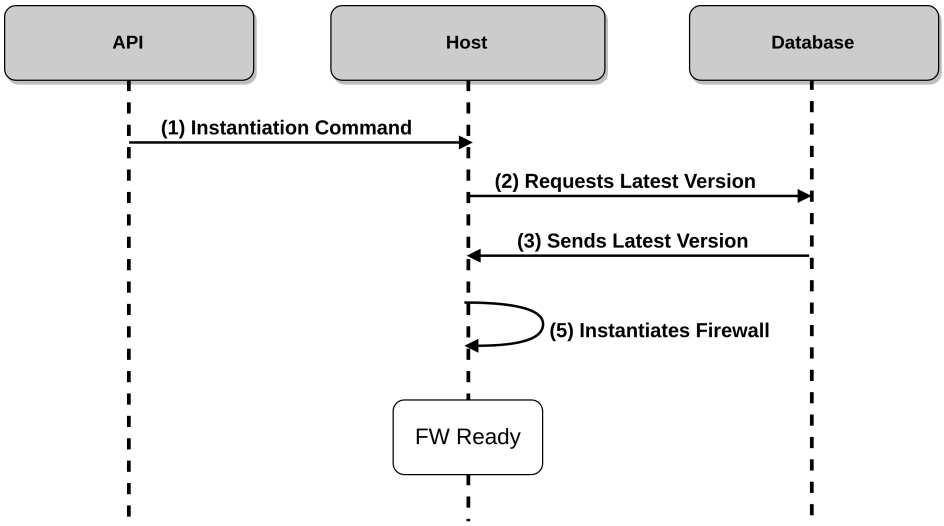

Figure 3.8: Stateful IncludeOS Firewall Instantiation Procedure

3.2.5.3 Firewall Behavior

Figure 3.9 shows two different scenarios where a hypothetical client tries to access the content of an HTTP server hosted in the sink node.

In scenario A the client makes a request to access the server that is running in an IP/Port that is on the whitelist specified in the firewall policies:

1. The request is sent to the host;
2. The br-dabc81852bad bridge forwards the packets to the firewall;
3. After checking the policies in action, the firewall routes the packets to the br-8c1fd3936e33 bridge that forwards them to the HTTP server;
4. The reply is sent from the server to the client in the reverse path.

In scenario B, contrary to A, the server is hosted in an IP/Port that is not included in the whitelist:

1. The request is sent to the host;
2. The packets are forward to the firewall by the br-dabc81852bad bridge;

The firewall checks the policies in action and since the IP/Port of the server is not to be accessible by the public network, the VNF immediately drops the packets.

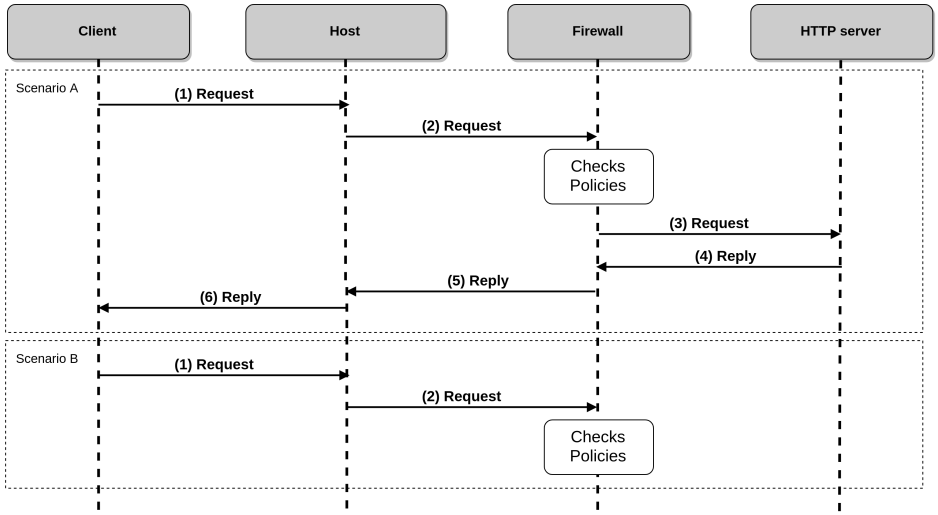

Figure 3.9: Firewall Behavior Diagram

3.2.6 Failure Detection Mechanism

As stated in section 3.1, the cloud provider needs to cope with the possibility of failure. For this reason, a failure detection mechanism was developed that verifies the status of operation of the firewall. This mechanism is based on the heartbeat software developed by the Linux-HA (High-Availability Linux)[10]. It was developed in bash and its behavior is described in Figure 3.10.

The failure mechanism starts by periodically sending ICMP requests. These requests are sent every 100 ms. If the VNF responds by sending an ICMP reply, the failure mechanism

[10]http://www.linux-ha.org/

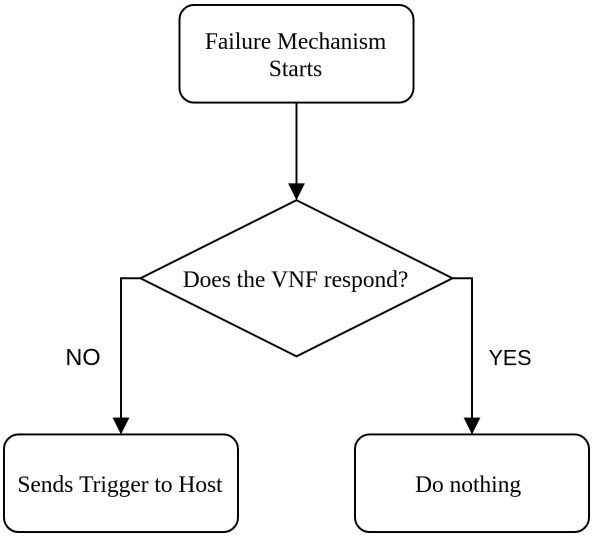

Figure 3.10: Failure Mechanism Behavior

does nothing and proceeds to keep sending ICMP requests. When the VNF fails to respond within a 50 ms time-frame, a trigger is sent to the host. After receiving the trigger message, the host kills the processes of the VNF so that the switching mechanism can re-instantiate a new firewall in the desired platform - container or unikernel. The platform in which the firewall is going to be re-instantiated is sent in the trigger message.

This failure mechanism is an illustrative approach with the sole purpose of triggering dynamic re-adjustments of the system. Other more refined solutions including monitoring and operation assessment mechanism more tailored towards commercial environments could be used, but they are not the focus of this work.

3.2.6.1 Switching Mechanism

This mechanism was developed to re-instantiate the VNF after receiving the trigger from the Failure Detection Mechanism. Additionally a functionality that allows the possibility of switching between technologies (containers or unikernels) was also implemented in this mechanism. The switching mechanism, which has its behavior represented in Figure 3.11, starts by verifying if a trigger sent by the failure detection mechanism was received. If this is true, the switching mechanism instantiates the VNF in the platform specified in the trigger message. If it is not a valid platform (unikernel or container) an error message is printed. On the other and, if it does not receive any trigger message, the mechanism reads the desired instantiation platform and, in the same way as stated above, if it is not a valid platform, a error message is printed. If the container platform is chosen, the program checks if the firewall is already running in a container. In the case that it is, then nothing is done. Otherwise, if the firewall is running in an unikernel, the host kills the processes of the firewall and re-launches the VNF as a docker container. Similarly, if the unikernel platform is chosen, the same logic is applied but now for the IncludeOS Firewall.

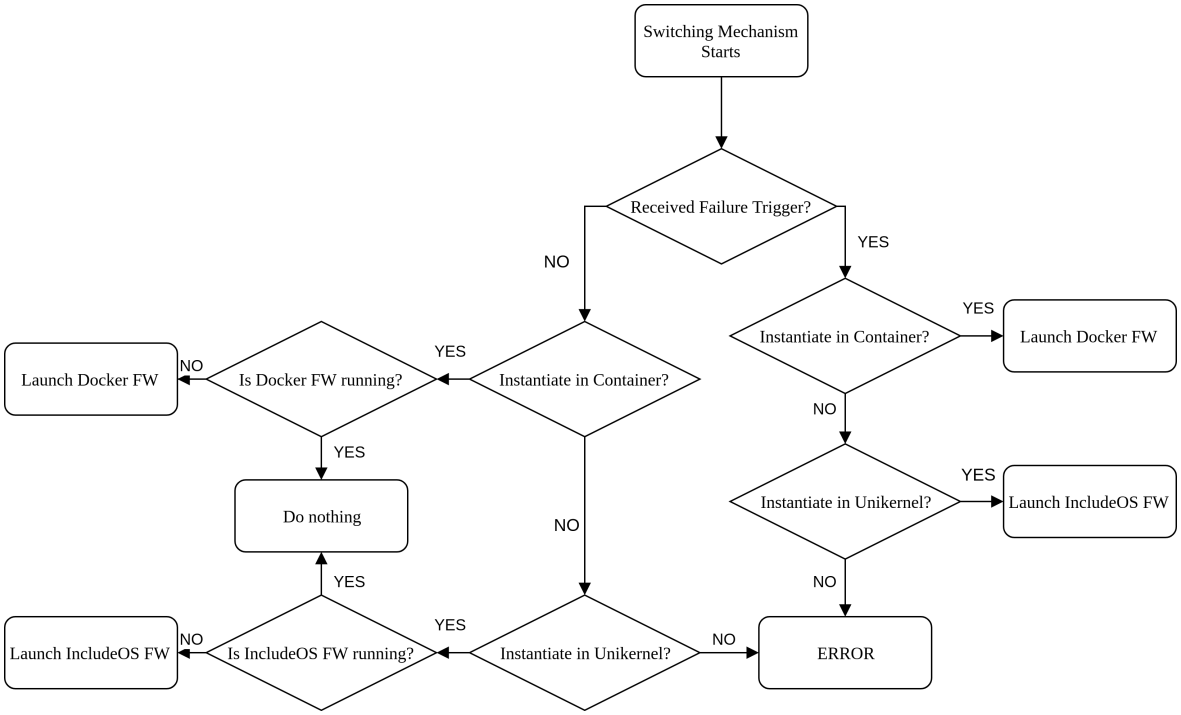

Figure 3.11: Switching Mechanism Behavior

3.2.6.1.1 Failure Detection and Re-instantiation Procedure.

Figure 3.12 presents the procedure of the two mechanisms working together to re-instante the firewall when a disruption occurs.

1. The failure detection mechanism sends ICMP requests to the firewall;
2. The firewall responds;
3. Another ICMP request is sent;
4. The VNF doesn't respond within the time-frame causing the failure mechanism to send the trigger message to the switching mechanism;
5. After reading the desired virtualization platform in the trigger message, the switching mechanism sends a command to the host machine to kill the process of the defective firewall;
6. Right after sending the message in (5), the switch program sends the instantiation command to the host.

When the hypervisor receives the command, the stateful instantiation procedure for the selected platform begins. This process is fully explained in the previous sections 3.2.5.1.1 and 3.2.5.2.1.

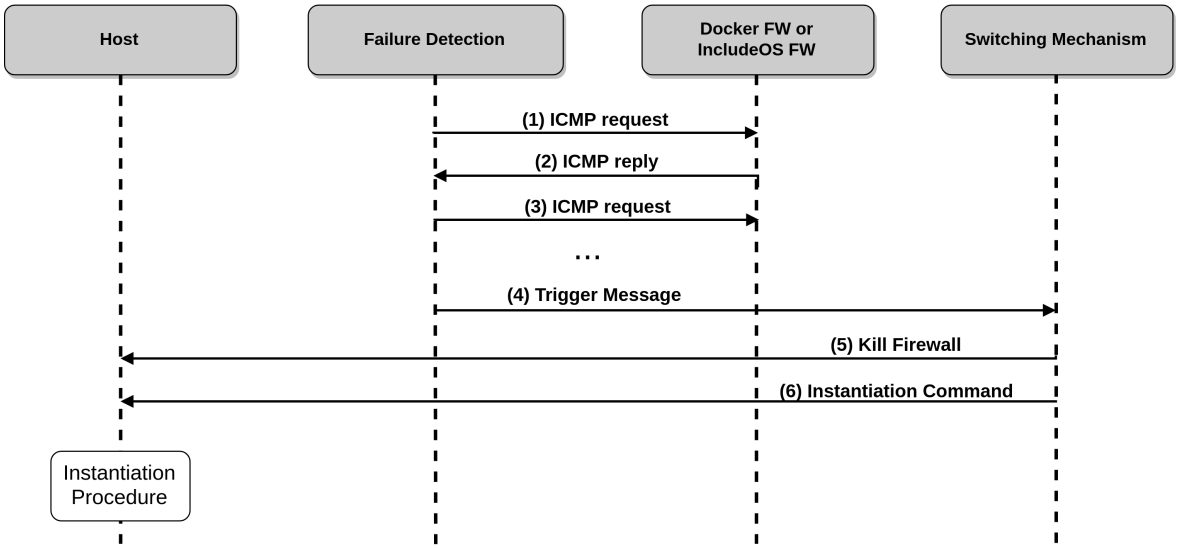

Figure 3.12: Firewall Failure with Re-instantiation Procedure

3.2.6.1.2 Switching Without Failure.

When the switching procedure is invoked without the VNF having failed, two options can happen: the firewall is not running in the desired platform and the host has to switch virtualization technology, or the firewall is already running in the selected technology. These two procedures are described in Figure 3.13 with option a) and option b), respectively.

1. A switch in the VNF platform is requested;
2. The switching mechanism verifies the technology in which the firewall is instantiated;

Option a)

3. The mechanism sends a command to the host machine to kill the process of the running VNF;

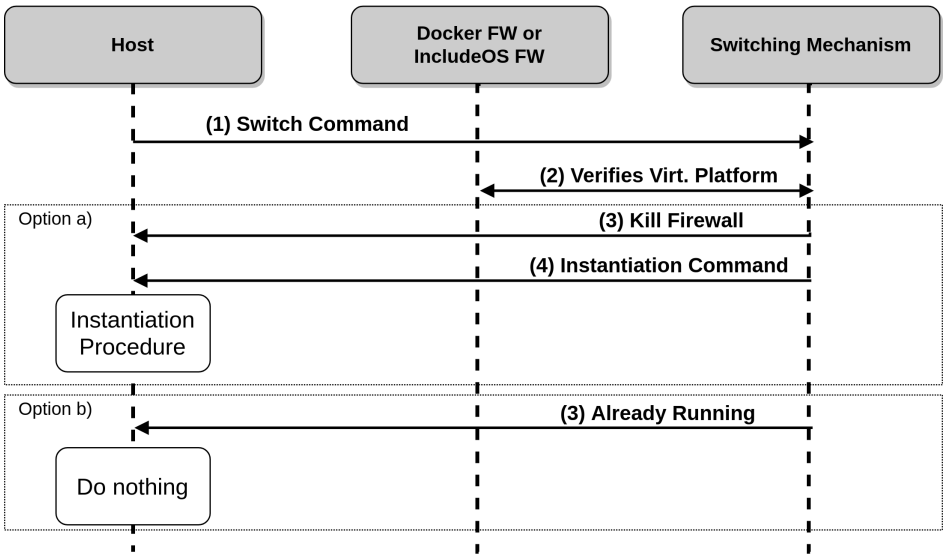

Figure 3.13: Switching Procedure

4. Right after sending the previous message, the switch program sends the instantiation command to the host.

After receiving the instantiation command the host starts the instantiation procedure in the opposite platform.

Option b)

3. The switch mechanism informs the host that the firewall is already running in the desired platform.

3.3 SDN Recovery Mechanism

To further reduce the downtime of the VNF, a SDN recovery mechanism was implemented. This mechanism prevents loss of connection between the outside network and the private network between a VNF fail detection and the re-instantiation of a new VNF, providing seamless VNF instantiation.

To achieve this, the Ryu SDN controller was installed and the Linux bridges - br-dabc81852bad and br-8c1fd3936e33 - were replaced with OvS bridges - br0 and br1 respectively. Both bridges are connected to the SDN controller. A full architecture is presented in Figure 3.14:

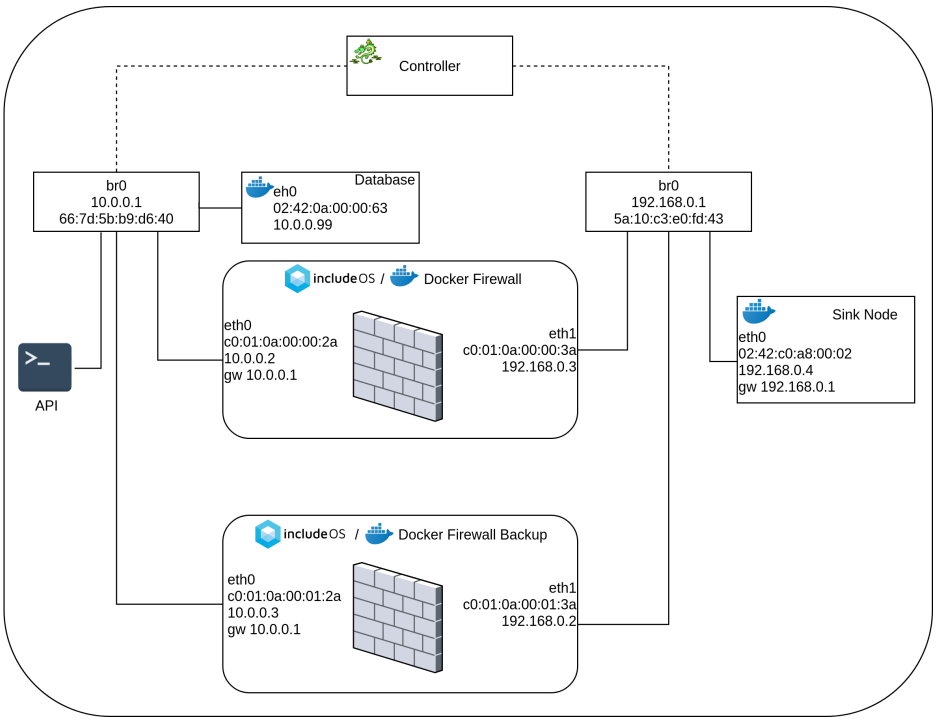

Figure 3.14: Detailed design with SDN Recovery Mechanism

To achieve this, a backup VNF is necessary, that can be automatically instantiated when the main VNF shows signs of failure (e.g. memory usage close to memory limit) or can be running in parallel with the main VNF.

Fig. 3.15 shows the diagram for two cases:

Case a) - A failure is detected in the firewall:

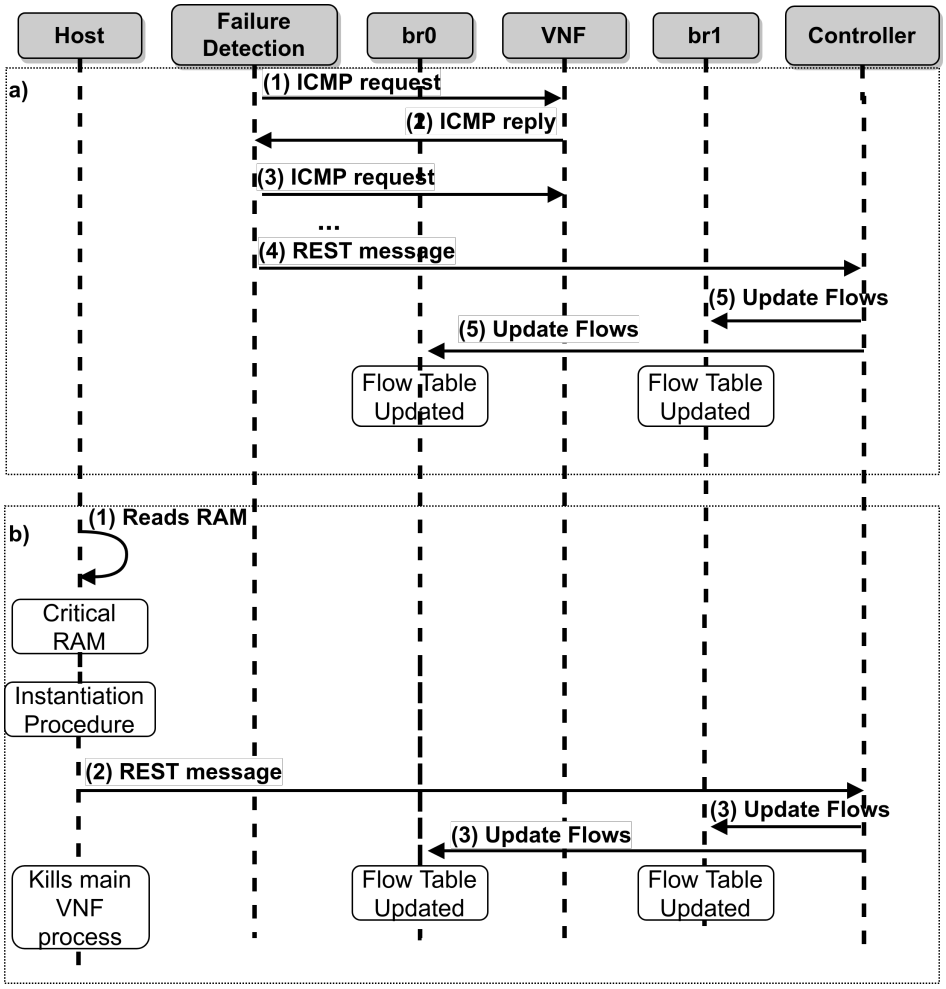

Figure 3.15: SDN Recovery Control Signaling Diagram

1. The failure detection mechanism sends Internet Control Message Protocol (ICMP) requests to the firewall;
2. The VNF responds with an ICMP reply;
3. Another ICMP request is sent;
4. Since the firewall does not respond to the ICMP request, meaning that an error occurred, a REST message is sent to the SDN controller, notifying the failure of the main VNF;
5. After receiving the trigger message, the controller updates the flow tables of both OvS bridges.

Case b) - Threshold of used RAM is reached triggering the instantiation of the backup VNF:

1. The host reads the RAM consumption of the main VNF;
   The RAM consumption reaches critical values (80% of total RAM allocated to the VNF), the instantiation procedure of the backup VNF begins;
2. When the backup VNF is fully operational, a REST message is sent to the SDN controller;
3. After receiving the trigger message, the controller updates the flow tables of both OvS bridges.

Once the OvS bridges update their flow tables, the packets that were previously being sent to the main VNF, are now forwarded to the backup VNF, ensuring the reliability of the service.

3.4 Summary

This chapter introduced an overview of the design for the critical and reliable use-case scenario. Both architectures, with container and with unikernel VNF were also presented, as well as the details for the remaining infrastructure and mechanisms to ensure reliability. In the following chapters the VNFs using both virtualization technologies are evaluated and the results are presented.

4 | Evaluation

This chapter presents the tests and relating performance results regarding the architectures using two types of virtualization presented in the previous chapter.
Both architectures were evaluated in several indicators such as image size, instantiation times, latency, throughput, jitter, TCP retransmissions and percentage of datagrams lost.

For the Docker and IncludeOS VNFs, all the tests were done using the architectures presented in Figures 3.3 and 3.6. These architectures are fully described in section 3.2 of chapter 3. Unless stated otherwise, all the tests were executed 10 times and the related results present their average values with a confidence interval of 95%.

4.1 VNF Size

Table 4.1 compares the image size of the two VNFs.

VNF	Size (MBytes)
Docker Firewall	16.4
IncludeOS Firewall	3.2

Table 4.1: VNF Image Size

It can be seen that the Docker firewall is approximately 5 times larger than the IncludeOS firewall. The reason for this huge difference has to do with the versatility of the VNF. The Docker Firewall runs as a tiny OS while the unikernel has limited capacities, having the sole purpose of running the C++ code compiled into the image. It should be noted that the actual size of the firewall program is also different for the two implementations, the Docker VNF uses the iptables package[1] (1.54 MBytes) and the IncludeOS firewall itself uses 0.5 Mbytes, since the unikernel platform needs a minimum of 2.7 MBytes to be able to run.

As for memory, both implementations have 256 MB of RAM allocated.

4.2 Instantiation Time

A crucial metric regarding critical and reliable communications is the time it takes to instantiate a VNF when requested. For measuring the instantiation of the two different VNFs, it was considered the time it took from the launch command until the firewall was ready with

[1] https://pkgs.alpinelinux.org/package/edge/main/x86/iptables

all policies implemented. This instantiation process was described in sections 3.2.5.1.1 and 3.2.5.2.1.
The time was measured immediately before the instantiation command and immediately after the last rule, measured in milliseconds using the Unix epoch time[2]. Figure 4.1 shows the instantiation time for both Docker and IncludeOS Firewalls with the two configurations, stateless and stateful.

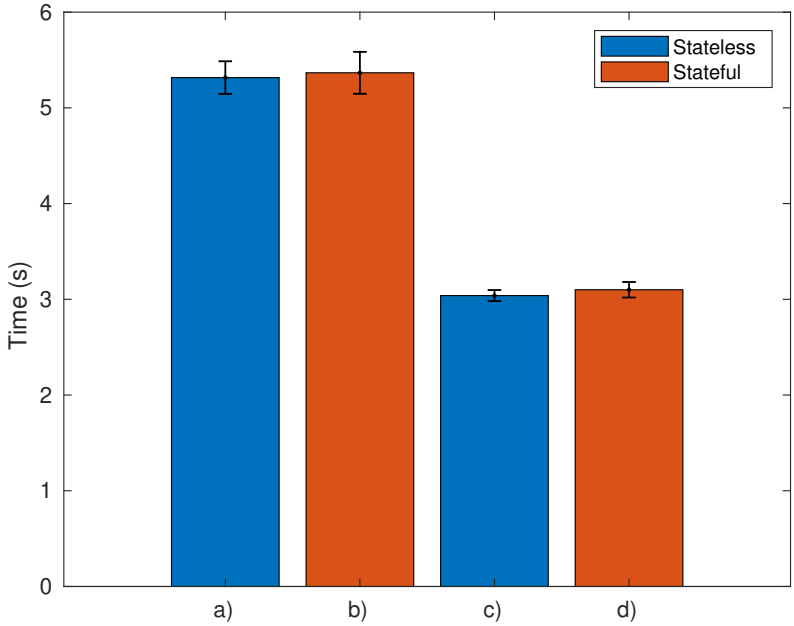

Figure 4.1: Instantiation Time - a) Stateless IncludeOS Firewall; b) Stateful IncludeOS Firewall; c) Stateless Docker Firewall; d) Stateful Docker Firewall

It can be clearly seen that when the VNF is instantiated in a container it's around 2,3 seconds faster than when it's launched in a unikernel. Regarding the difference between the stateless and stateful configuration, for both virtualization platforms, the instantiation takes approximately the same amount of time. The location of the Database container has to be taken into account for the stateful instantiation. In this architecture the Database is in the same location as the VNF. If this Database was located in another data-center for example, the launching time would have increased since the firewall version would have taken more time to download.

4.3 Latency

For the purpose of this evaluation, the latency refers to the Round Trip Time (RTT) of a packet. This test intends to see how the VNFs affect the latency in a end-to-end connection and was measured using the ping tool, belonging to the Network Tools Package[3]. This tool

[2]https://www.epochconverter.com/
[3]https://network-tools.com/

generates an ICMP Request every second and waits for a reply, measuring the time between the two. This test was done 10 times from the side of the API and 10 from the side of the Sink node - Figures 3.3 and 3.6 show, in bold lines, the travel path for the ICMP packets. The results are presented in Figure 4.2.

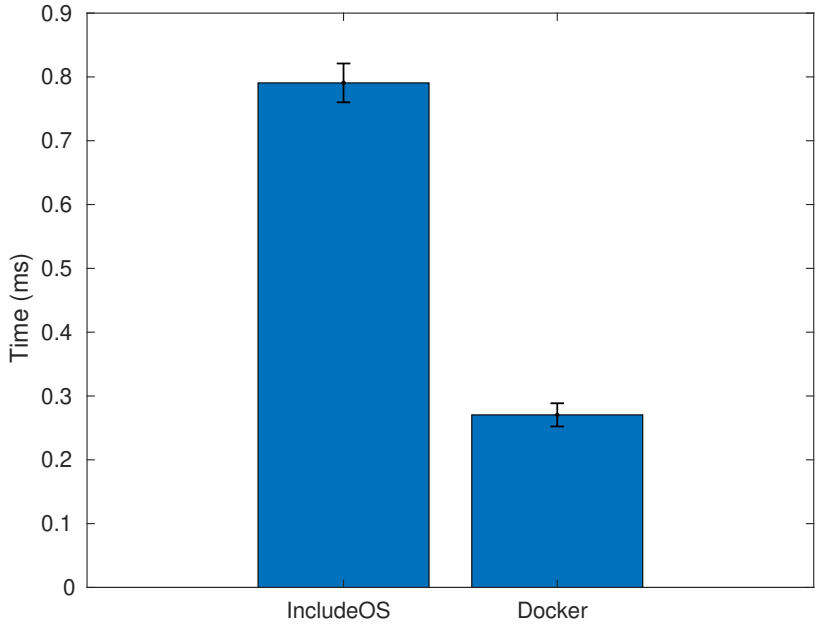

Figure 4.2: Latency

From observing the figure, the latency of the end-to-end connection when the firewall is instantiated in an unikernel is approximately 0.79 ms, as for the Docker instantiation, end-to-end latencies round the 0.27 ms, almost 3 times less. It can be deducted that this latency difference happens because of the time the packets are being processed by the firewall, since the ICMP packets travel the same path in both implementations.

4.4 Policy Updates

This section focuses on the evaluation of the VNFs when submitted to policy updates. The two technologies are updated in different ways due to the nature of the each firewall.

To perform updates on the IncludeOS firewall, the LiveUpdate software[4] developed by Alf-Andre Walla was used. LiveUpdate is a state-transfer based Dynamic Software Updating (DSU), meaning it stores only the state, the minimal amount of data, that service writers deem necessary to resume service operation seamlessly after an update [47]. With this feature, the unikernel updates when a new version of the binary is sent to the platform.

For the Docker firewall to update its rules, a bash script with new iptables rules is sent to execute in the container[5]. After the script is executed, the firewall has the new policies

[4] https://github.com/hioa-cs/IncludeOS/tree/master/examples/LiveUpdate
[5] https://docs.docker.com/engine/reference/commandline/exec/

applied.

The next subsections present the results of three different scenarios of policy updates for the two firewalls.

4.4.1 Whitelist Static Update

In this first scenario the firewall whitelist is updated. Initially the VNFs only have 3 IP addresses to which is allowed connection from the public network (192.168.0.1-3). After the whitelist is updated, the firewall allows connections from 100 addresses in the private network (192.168.0.1-100). Figure 4.3 shows the update time for the Docker Firewall and for the IncludeOS Firewall when the new rules need to be compiled into the new binary and when the binary is previously compiled. The tests were performed 20 times and present the mean value and a confidence interval of 95%.

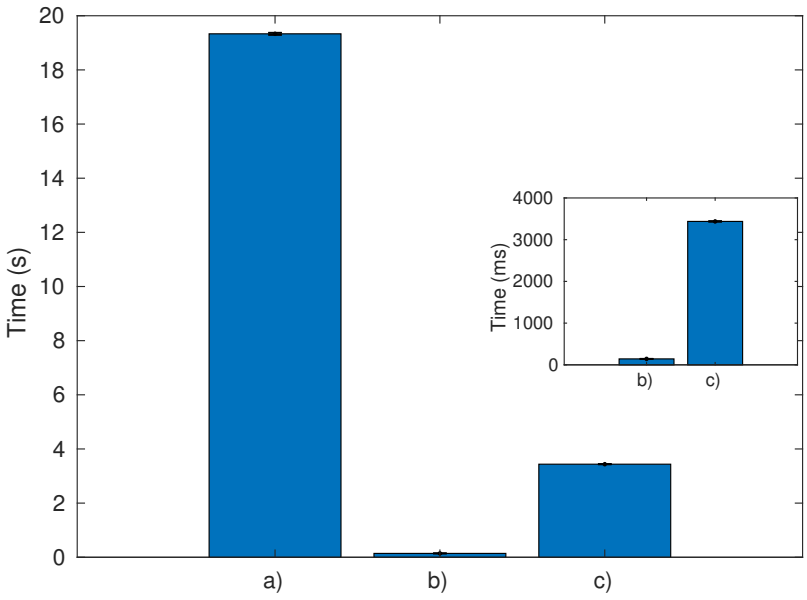

Figure 4.3: Whitelist Static Update - a) IncludeOS FW With Compilation; b) IncludeOS FW Pre-Compiled; c) Docker FW

It is clear that when the IncludeOS firewall needs to be compiled on demand, the update times are enormous compared to the other two, as most of the time is spent by the C++ compiler. On the other hand, when the binary is previously compiled, the IncludeOS update is the fastest taking approximately 150 ms which is around 23 times faster than the Docker Firewall.

4.4.2 Whitelist Incremental Update

The second scenario is very similar to the first one, but instead of statically updating the whitelist from 3 to 100 allowed addresses, the whitelist is updated with increments of 10, starting with 10 addresses and finishing with 254 addresses. For all updates the baseline

version only has 3 IPs in the whitelist. The tests were made 3 times for each incrementation and the mean values are presented in Figure 4.4.

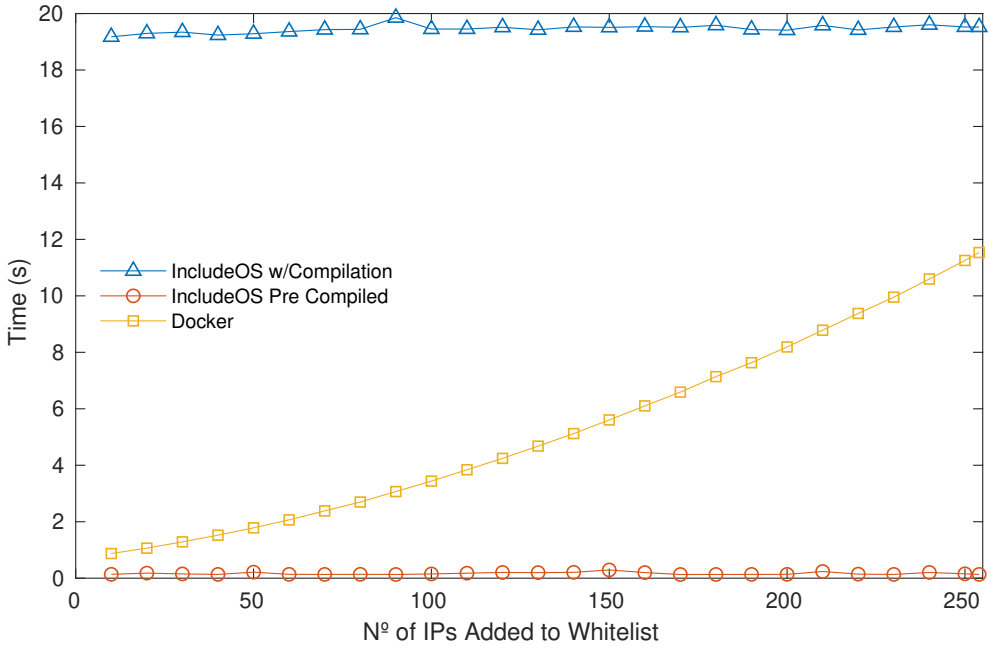

Figure 4.4: Whitelist Incremental Update

From evaluating the previous figure, two aspects are evident. The first is that the number of IPs added to the whitelist doesn't affect the time of compilation or the update time for IncludeOS. This behavior was expected as the C++ code needs to be modified and compiled before the VNF can be launched with new policies.

For second, it is seen that the update time for the Docker Firewall exhibits a slow exponential rate ranging from around 1 to 12 seconds. This happens because the script updating the policies needs to apply the iptables commands to a large number of IPs, taking a longer amount of time.

4.4.3 New Rule Update

In this last scenario, a new rule is added to the firewall policy - the port 444 is opened for TCP traffic. Similarly to the two previous subsections, the VNFs have the whitelist with only 3 IPs. The tests were performed 20 times and the Figure 4.5 presents the mean value and a confidence interval of 95%.

As in the previous scenarios, the IncludeOS firewall update is the fastest when pre-compiled. Theres is a slight decrease of nearly 2.5 seconds in update time of the Docker container in comparison to the first scenario and it takes approximately the same time as the first sample in the second scenario.

From this three scenarios examined above, it became evident that the update time of the IncludeOS Firewall remains the same for every case either with compilation or pre-compiled -

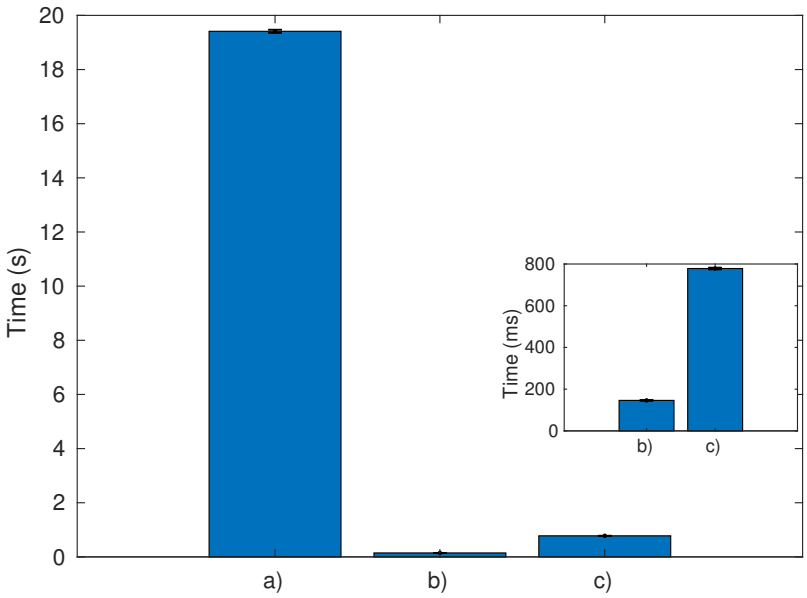

Figure 4.5: New Rule Update - a) IncludeOS FW With Compilation; b) IncludeOS FW Pre-Compiled; c) Docker FW

roughly 150 ms. As for the Docker Firewall, the update time varies according to the number of iptables commands in the update script, being approximately 780 ms the lowest recorded update time.

In the following section, the VNFs will be tested to see how they perform when traffic passes through as well as when a update occurs.

4.5 Traffic Traversing VNF

This section addresses the overall ability to handle high quantities of traffic passing through the firewall. Three performance tests were made for each VNF, the first with TCP traffic and the other two with UDP traffic - 100 Mbps target bandwidth and 1 Gbps target bandwidth. These tests were performed using the iPerf3 tool[6]. The iPerf server was started on port 80 of the Sink container. The client makes connection from the CLI of the data-center with a buffer length of 1400 Bytes.

4.5.1 TCP Traffic

Figure 4.6 shows the results after performing the iPerf 20 times for each platform using TCP packets.

The Docker Firewall has a better performance in both metrics. An average throughput of 120 Mbits/s compared to 100 MBytes/s from the IncludeOS and no TCP segments were retransmitted in all 20 tests as in the unikernel there were an average of 1.10 TCP segments retransmitted, that translates into 0.013% of the total TCP packets sent. Notwithstanding

[6]https://iperf.fr/

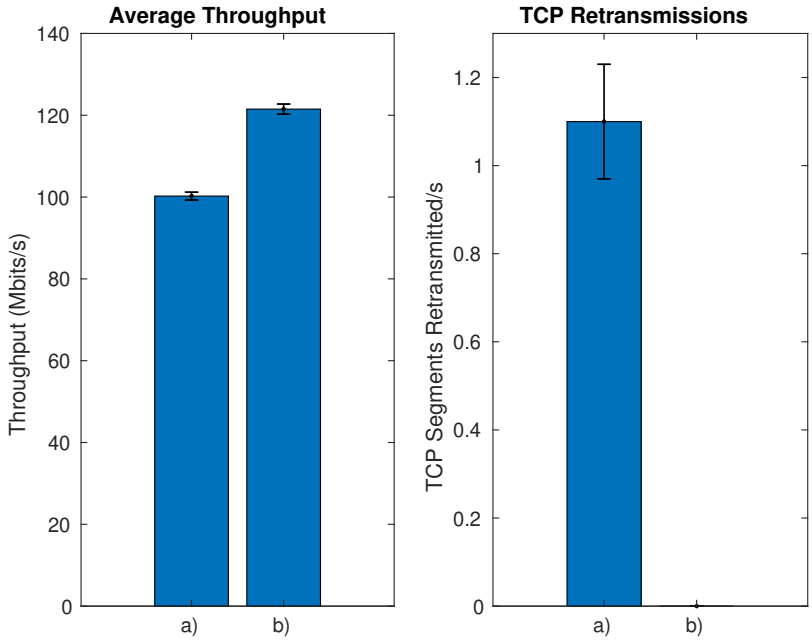

Figure 4.6: TCP Performance - a) IncludeOS FW; b) Docker FW

the small percentage of retransmitted packets, this difference can be justified by the latency delay introduced by the IncludeOS firewall, as it was shown in section 4.3.

4.5.2 UDP Traffic

For UDP traffic two target bandwidths were set - 100 Mbits/s and 1Gbit/s - represented in blue and red respectively in Figure 4.7. Like the previous subsection, the tests were performed 20 times for each VNF.

Contrary to 4.5.1, IncludeOS has approximately the same performance as the Docker with 100 Mbits/s target bandwidth, only stays behind in the datagram loss since the container doesn't lose any datagrams.

When the target bandwidth is set to 1 Gbits/s on the iPerf client, the performance of the unikernel improves in comparison to the container, 160 Mbits/s of throughput - 20 Mbits/s better than the container.

In both technologies the target bandwidth of 1 Gbps is not reached due to the size of the UDP packets (1400 bytes).

In contrast, the average Jitter in the Docker Firewall stays constant and in the unikernel increases to double (0.06 ms). The datagram loss stays equal from the previous setup - 1% for IncludeOS and zero losses for Docker. This losses in the unikernel happen for the same reason as the TCP retransmissions in 4.5.1, the delay introduced by the IncludeOS Firewall.

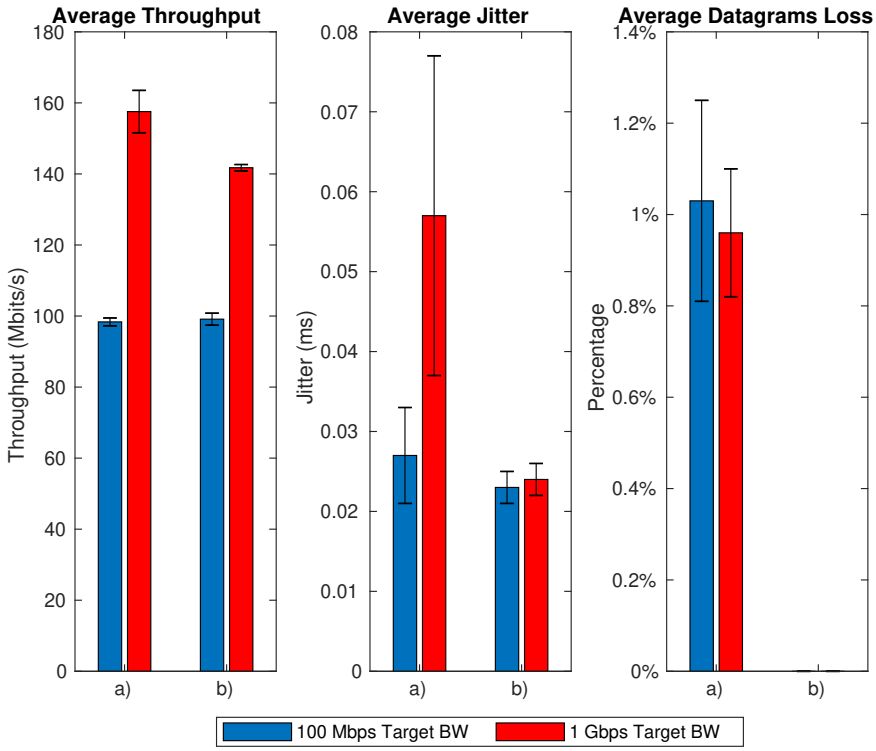

Figure 4.7: UDP Performance - a) IncludeOS FW; b) Docker FW

4.5.3 TCP Traffic with Firewall Update

To evaluate the impact of an update on the VNFs when traffic is passing through, an update was launched into the firewall while the iPerf is running. This iPerf has the same specifications as in 4.5.1 and was performed 10 times during 10 seconds, with the results being presented in the following Figure 4.8.

For both VNFs, the update is sent at Time = 2 sec.
For the IncludeOS firewall, represented in blue, the update can be easily seen happening at Time = 5 sec. where there is a decrease in throughput. This decrease is followed by a high increase in TCP segments being retransmitted in the following seconds.

Regarding the Docker VNF, the update can also be seen at Time = 6 sec. where a decrease also occurs but less pronounced. In comparison to the unikernel, the container doesn't retransmit any TCP segments meaning that there weren't any failures in transmission. This major difference in TCP retransmission between the two platforms happens because the IncludeOS firewall is disabled for the time it takes to update, while the Docker firewall doesn't need to be disabled to be able to update.

For both implementations, the update takes longer than the estimated time presented in the previous section 4.4. This difference happens due to the high quantities of traffic passing through the VNF.

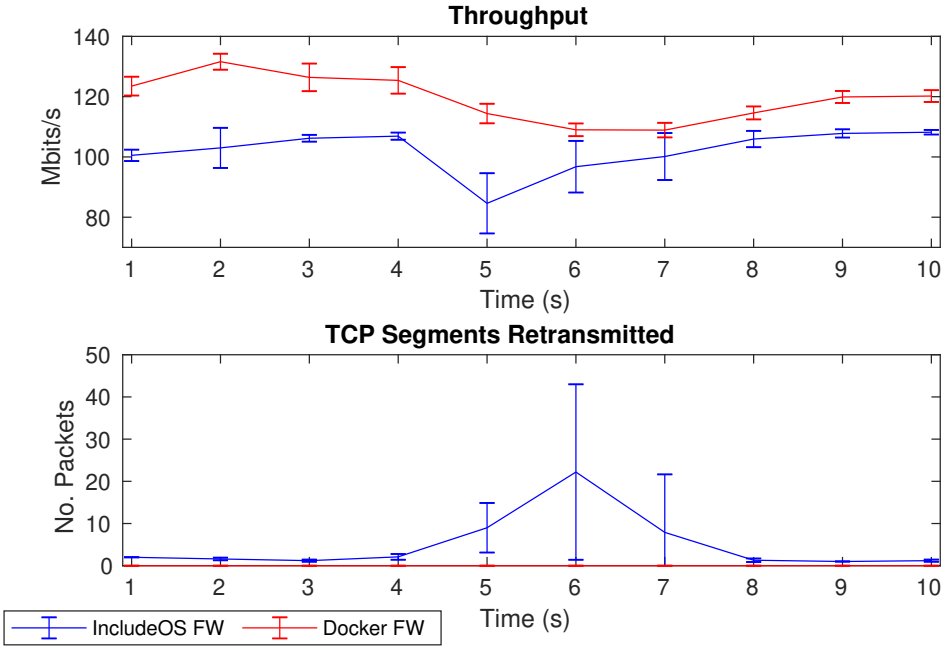

Figure 4.8: TCP Performance with Firewall Update

4.5.4 UDP Traffic with Firewall Update

In the same way as in 4.5.2, both types of virtualization were tested with two different iPerfs for UDP traffic, the first with 100 Mbps target bandwidth and the second with 1 Gbps. For both bandwidth targets, the update was sent at Time = 2 sec.

4.5.4.1 100 Mbps Target Bandwidth

The results of the 10 performance tests are presented in Figure 4.9. After evaluating the results, the IncludeOS firewall, in the same manner as in 4.5.3, at Time = 6 sec. a decrease in throughput take place when the update occurs as well as an increase in the percentage of datagrams lost.

For the Docker container, the update happens almost unnoticed, since the percentage of datagrams lost is always zero, and the throughput remains in average values throughout the 10 seconds of evaluation time. Contrary to the IncludeOS firewall, with the container instantiation there is not disruption of the service since the Docker firewall keeps running while the rules are updated. The only sign we get from the update is a minor increase in Jitter at Time = 5 sec.

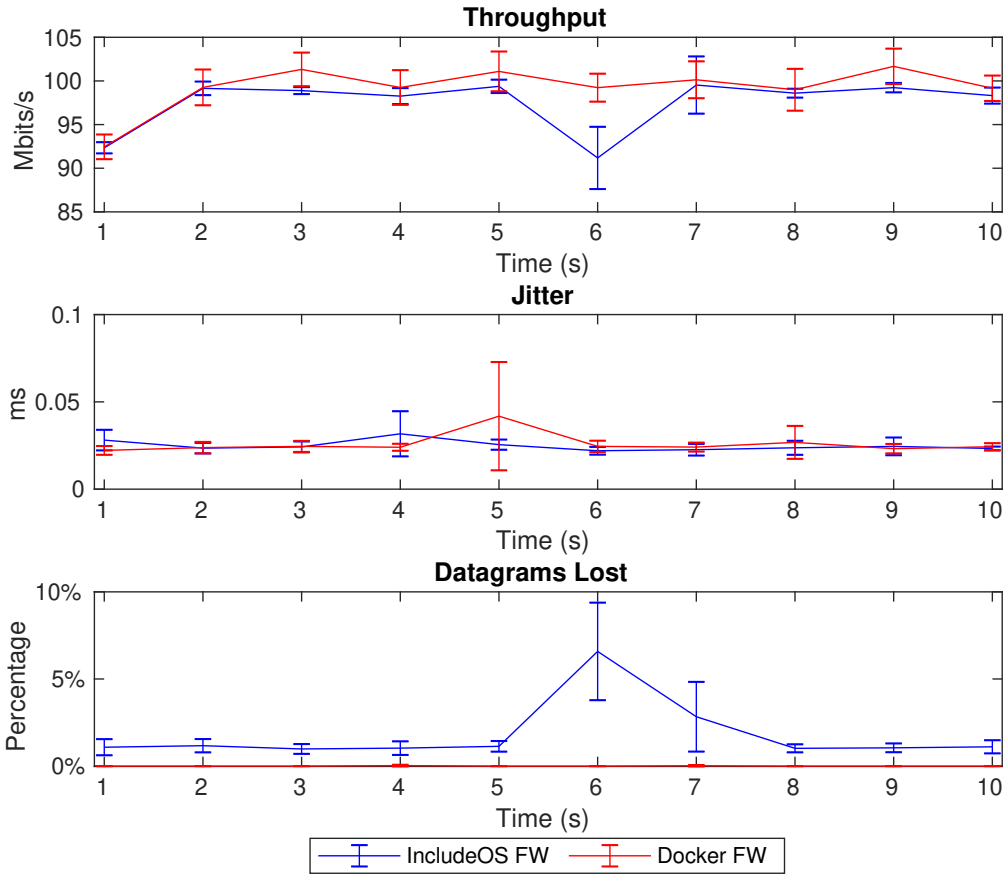

Figure 4.9: UDP (100 Mbps) Performance with Firewall Update

4.5.4.2 1 Gbps Target Bandwidth

When the target bandwidth is set to 1 Gbps in the iPerf, despite having a throughput decline when an update occurs (Time = 8 sec.), the IncludeOS firewall has better performance in comparison to the Docker firewall in this metric. The unikernel only falls behind in Jitter and in the number of datagrams lost where there is 80% of datagrams lost at Time = 8 sec. The overall evaluation for this test is explicit in Figure 4.10.

Overall, when the VNF is instantiated in a Docker container, its performance remains practically the same throughout the whole evaluation.

For the IncludeOS implementation, the throughput always declines when an update occurs for all three tests. It also exhibits approximately 80% of datagrams lost when the update is done while UDP 1 Gbps traffic is traversing the firewall.

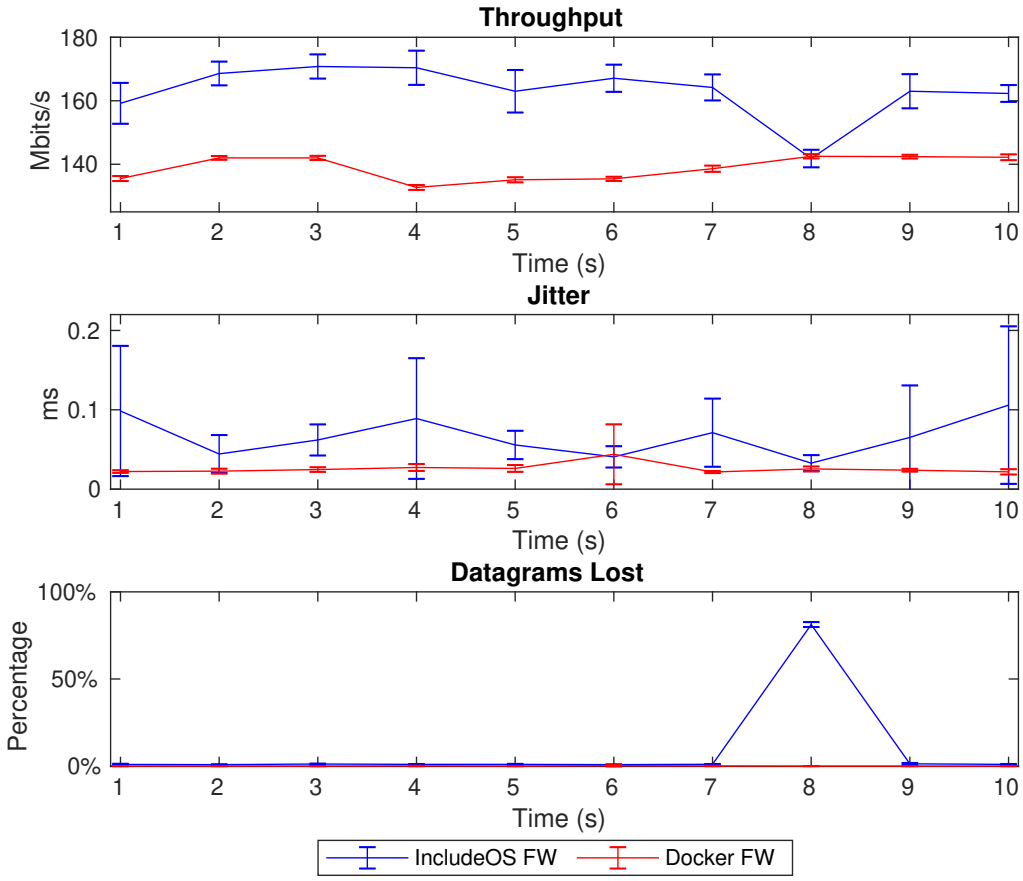

Figure 4.10: UDP (1 Gbps) Performance with Firewall Update

4.6 Failure Detection Mechanism

This section includes the tests and evaluation of both failure detection and switching mechanisms detailed in chapter 3.

4.6.1 Failure Detection Time

The failure detection time is determined based on time interval that the VNF fails to respond. This time interval is calculated using the Failure Detection Mechanism, described in section 3.2.6. To calculate this failure detection time, a time stamp was stored in a variable - t1 - before every ICMP request. When the VNF fails to respond, the trigger is activated and another time stamp is stored in a second variable - t2. Subtracting these two variables gives us an estimate of the time it takes to detect a failure in the firewall. This test was repeated 10 times and the average values for each VNF as well as the confidence intervals are presented in Figure 4.11:

From analyzing the figure, it is visible that the impact of the VNF is negligible for the time it takes the mechanism to detect the failure, having just a difference of roughly 3 ms.

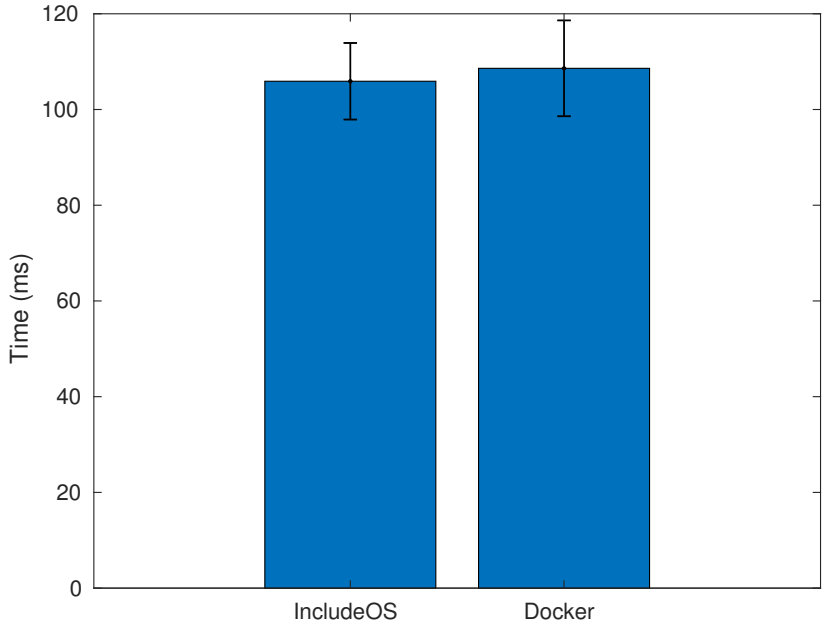

Figure 4.11: Failure Detection Time

4.6.2 Re-instantiation Time

For measuring the time of re-instantiation, the two reliability mechanisms are necessary - the Failure Detection Mechanism and the Swicthing Mechanism, both described in sections 3.2.6 and 3.2.6.1 of chapter 3. The re-instantiation time takes into account the time of failure plus the time that the switching mechanism takes to instantiate the VNF in the desired platform. Similarly to the previous test, the time stamp taken before the ICMP requests is stored in a variable and passed to the switching mechanism alongside the disered platform. After the re-instantiation of the firewall is completed, another time stamp is taken and subtracted to the time stamp from the failure mechanism given us the total time for the re-instantiation of the VNF.

Figure 4.12 presents the results for the 4 possible scenarios of re-instantiation: a) When the VNF is running as a IncludeOS unikernel and is re-instantiated in the same platform; b) The firewall is running as a Docker container and and is re-instantiated in the same platform; c) The VNF is originally instantiated as a unikernel and is recovered into a container; d) For theoretical purposes, a final scenario was also evaluated, where the firewall in running as a Docker container and is re-instantiated in a unikernel. This scenario is purely conceptual since the instantiation in a IncludeOS unikernel takes roughly 2 seconds longer than the re-instantiation in a Docker container, making it a less ideal solution.

From the figure it is clearly seen that the re-instantiation is faster when the VNF is re-launched into a Docker container, taking around 3 seconds in b) and 4 seconds in c). As for IncludeOS, it has the worst re-instantiation times, 5.5 seconds for a) and approximately 6.4 seconds for d).

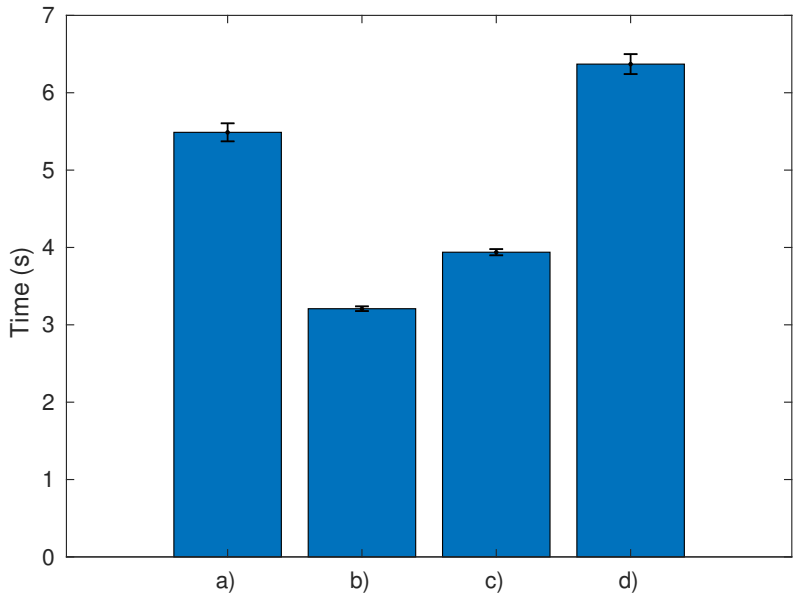

Figure 4.12: Re-instantiation Time - a) IncludeOS to IncludeOS; b) Docker to Docker; c) IncludeOS to Docker; d) Docker to IncludeOS

These two mechanisms will also be tested when a on-going connection is traversing the VNF, in the following subsection.

4.6.3 Traffic Behavior With VNF Failure

The following tests intend to evaluate the two reliability mechanisms in a realistic scenario and endorse the previous results.

The same iPerf tool as in 4.5 was used for simulating a connection between the API and the sink node. Both VNFs were tested with TCP and UDP traffic passing through.

4.6.3.1 TCP Traffic

For each VNF, the test was replicated 3 times, each time 10 evaluations of 10 seconds were made. At first, a failure was induced with the reliability mechanisms turned off. The second tests were done with the failure mechanism working and being the IncludeOS the desired re-instantiation platform. Finally the tests were repeated with the reliability mechanisms working and the re-instantiation to be done with Docker.

The results for all the tests are presented in the following figures. Figure 4.13 shows the results when a failure is induced in the IncludeOS firewall and Figure 4.14 present the results when the VNF is initially running as a Docker container.

For all the tests, the a failure was induced at Time = 3 sec.

From the evaluation of the two figures it is clear that when the reliability mechanism is not active, after the failure is induced, the throughput drops to zero at Time = 4 sec.

When the reliability mechanism is recovering to IncludeOS, the re-instantiation begins at Time = 9 sec. when throughput begins to return to its initial value.

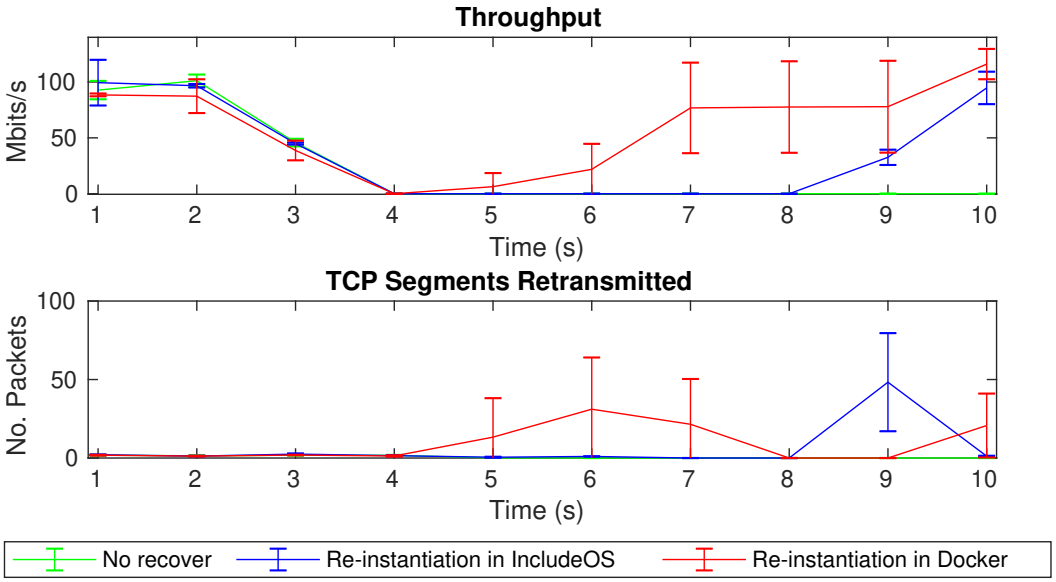

Figure 4.13: TCP Performance with IncludeOS Firewall Failure

Selecting the Docker VNF for re-instantiation, decreases the downtime of the firewall by 2 seconds, since throughput values are fully restored at Time = 7 sec.

The major differences between the two re-instantiation times was already expected since the same differences were acknowledge in the previous subsection.

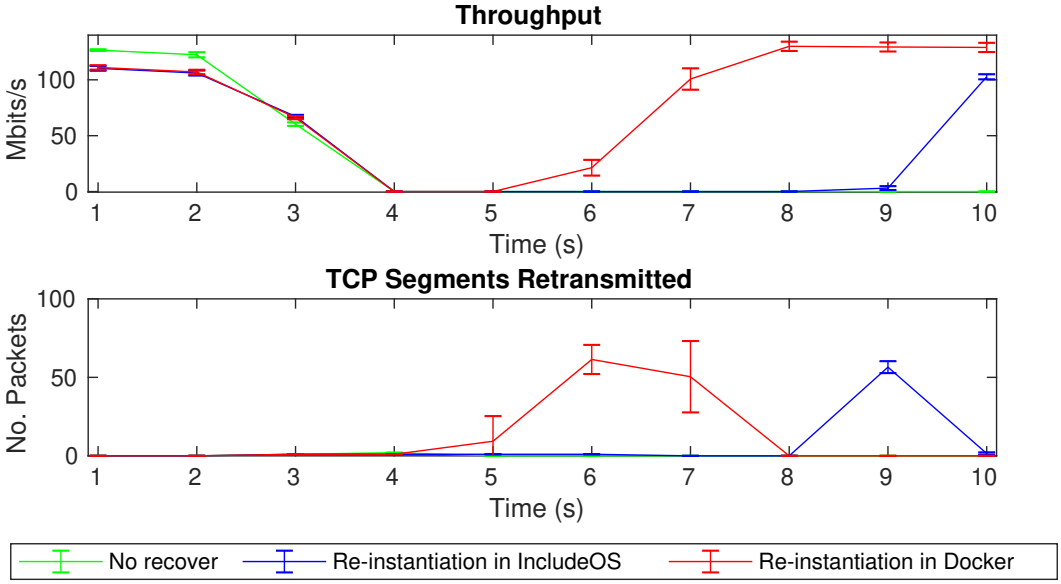

Figure 4.14: TCP Performance with Docker Firewall Failure

4.6.3.2 UDP Traffic

Regarding UDP traffic two types of iPerf were performed - 100 Mbps and 1 Gbps target bandwidth - as it was done in the previous sections. The same type of evaluation that was performed for TCP was now made for UDP.

The tests results are presented in the following figures. Figures 4.15 and 4.17 show the results when the fail is induced in the IncludeOS firewall, and Figures 4.16 and 4.18 present the results when the VNF is originally instantiated in the Docker platform. As in 4.6.3.1, the failure was induced at Time = 3 sec.

As in the previous experiments, when the recover mechanism is not active after the firewall disruption, throughput drops to zero and datagram loss rises to 100% at Time = 4 sec.

When the selected re-instantiation is IncludeOS, re-launching takes up to 6 seconds, only being fully restored at Time = 10 sec. having full loss of UDP packets during that time.

Docker continues to be the best platform for re-instantiation for both target bandwidths, as it can fully re-launch the service at Time = 7 sec., 3 seconds faster than the unikernel, and having just 3 seconds of full datagram losses.

From evaluating the figures from sections 4.6.2 and 4.6.3, it can be seen that the reliability mechanism behaves the same way when there is not traffic transversing the firewall and when the VNFs are processing high quantities of traffic, making it traffic independent.

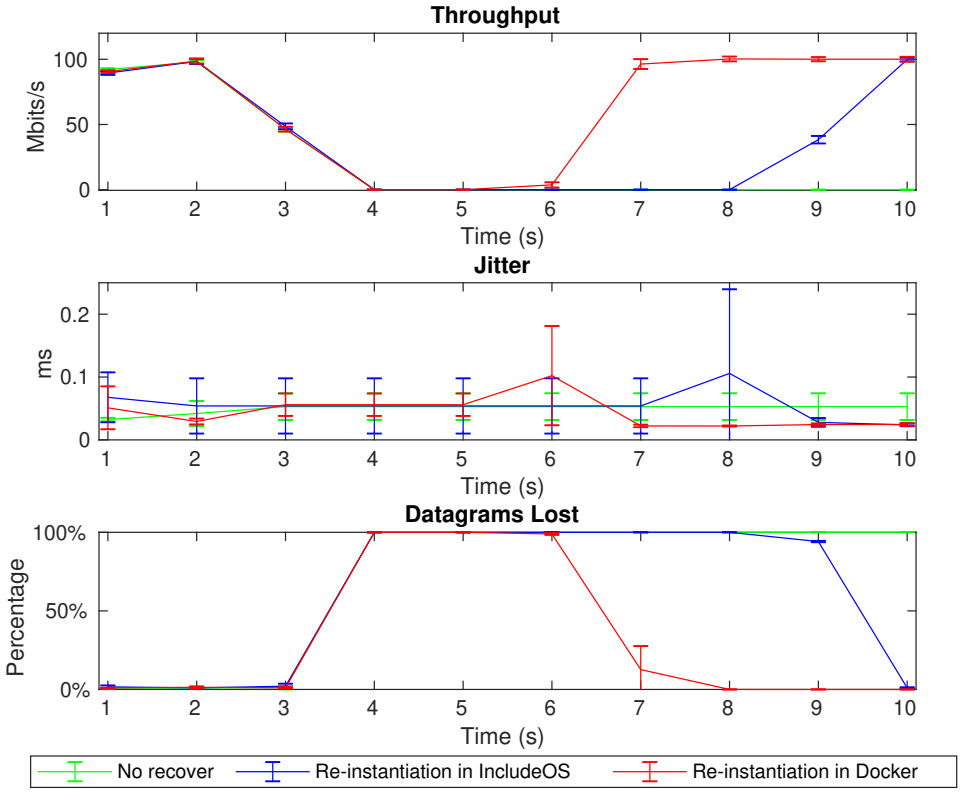

Figure 4.15: UDP (100 Mbps) Performance with IncludeOS Firewall Failure

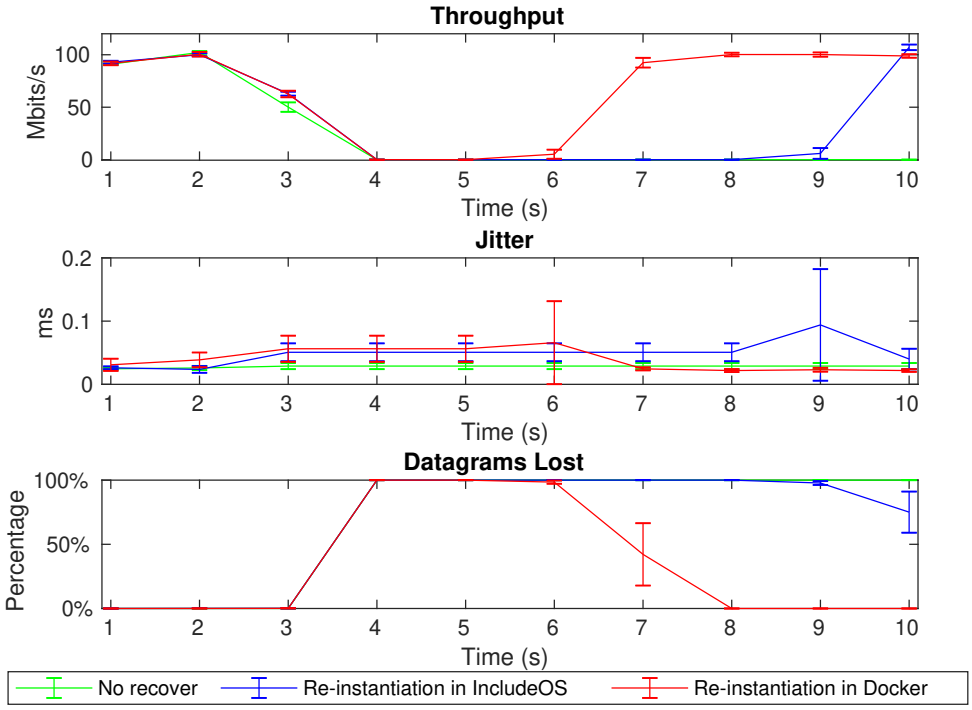

Figure 4.16: UDP (100 Mbps) Performance with Docker Firewall Failure

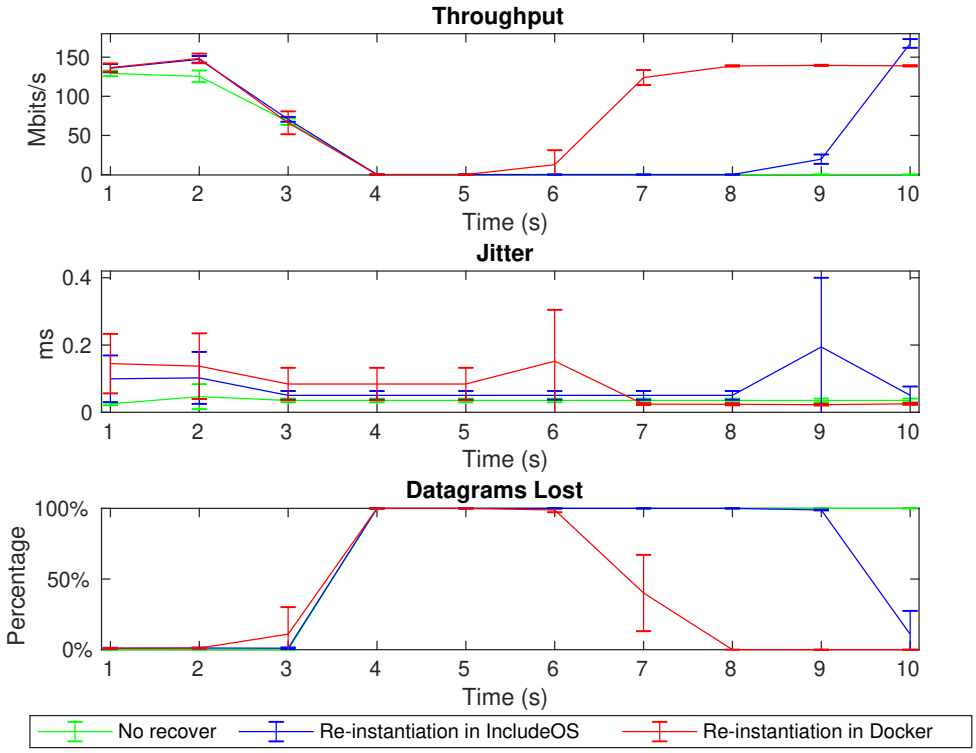

Figure 4.17: UDP (1 Gbps) Performance with IncludeOS Firewall Failure

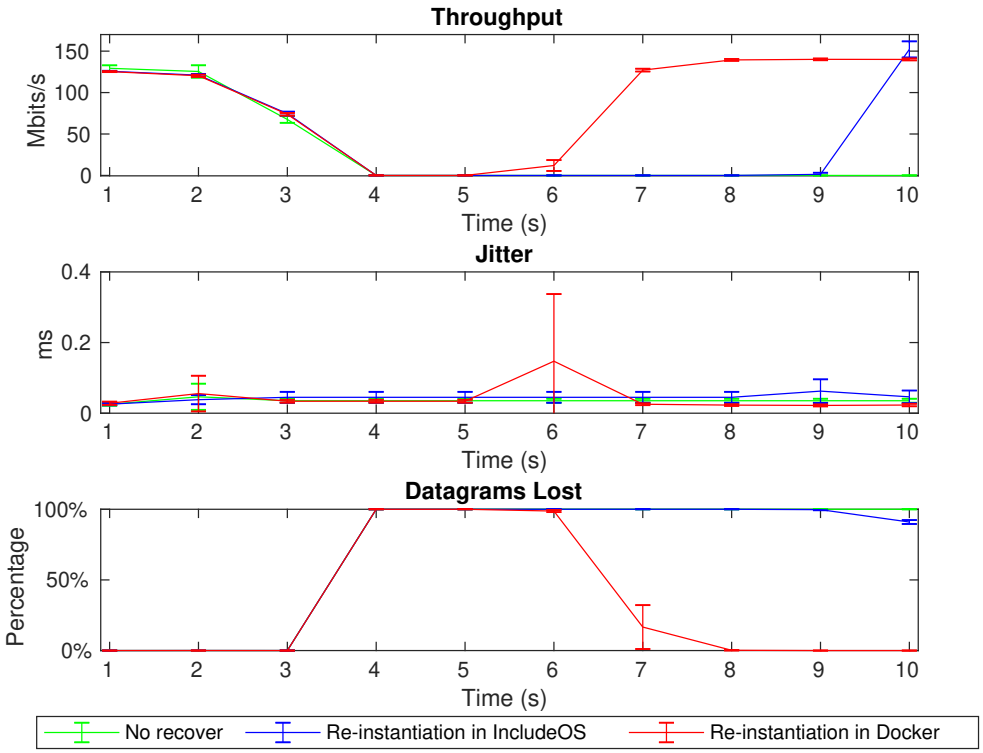

Figure 4.18: UDP (1 Gbps) Performance with Docker Firewall Failure

4.7 SDN Recovery Mechanism

To evaluate the reliability mechanism proposed in 3.3, the same iPerf tool as in 4.6.3 was used. In 4.7.1 the mechanism is tested with two instances running in parallel and in 4.7.2 the auto-instantiation procedure is evaluated.

4.7.1 Backup VNF Running in Parallel

As in 4.6.3 a failure is induced at Time = 3 sec. The results with TCP traffic are presented in Figure 4.19.

It is clear that the SDN Recovery Mechanism significantly improves the service as there is only a slight decrease in throughput when the flows are updated in the OvS bridges, this is evident at Time = 4 sec.

Similarly to the previous test, a failure is induced at Time = 3 sec.. The results, when the iPerf is performed with UDP packets with 100 Mbps and with 1 Gbps target bandwidth, are presented in Figures 4.20 and 4.21 respectively.

When the SDN Recovery Mechanism is active, for 100 Mbps target bandwidth, the redirection of traffic to the backup VNF is almost unnoticeable, with only a slight increase in Datagram loss seen at Time = 4 sec.

As for UDP traffic with 1 Gbps target bandwidth, the redirection of traffic to the backup FW is clear at Time = 4 sec., where a slight decrease of throughput is noticeable as well as roughly 25% of UDP datagrams are lost.

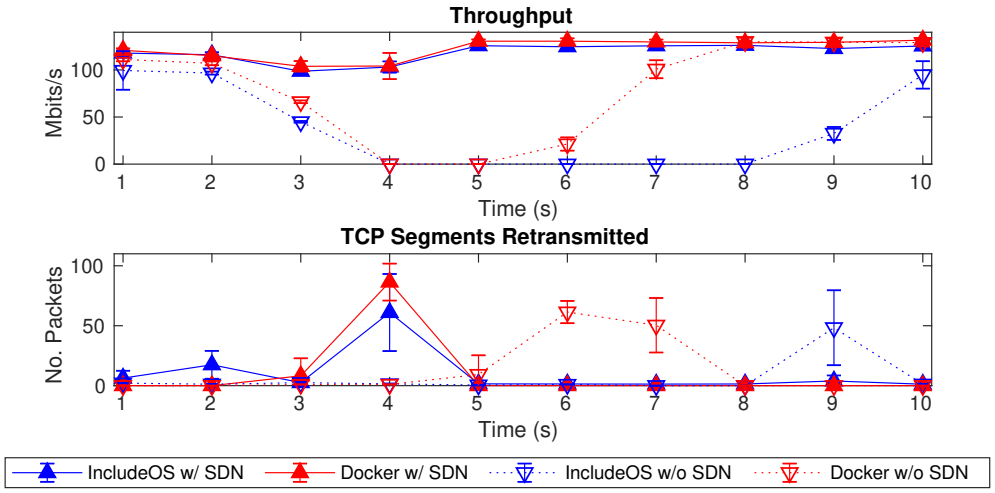

Figure 4.19: TCP Performance w/SDN Recovery Mechanism

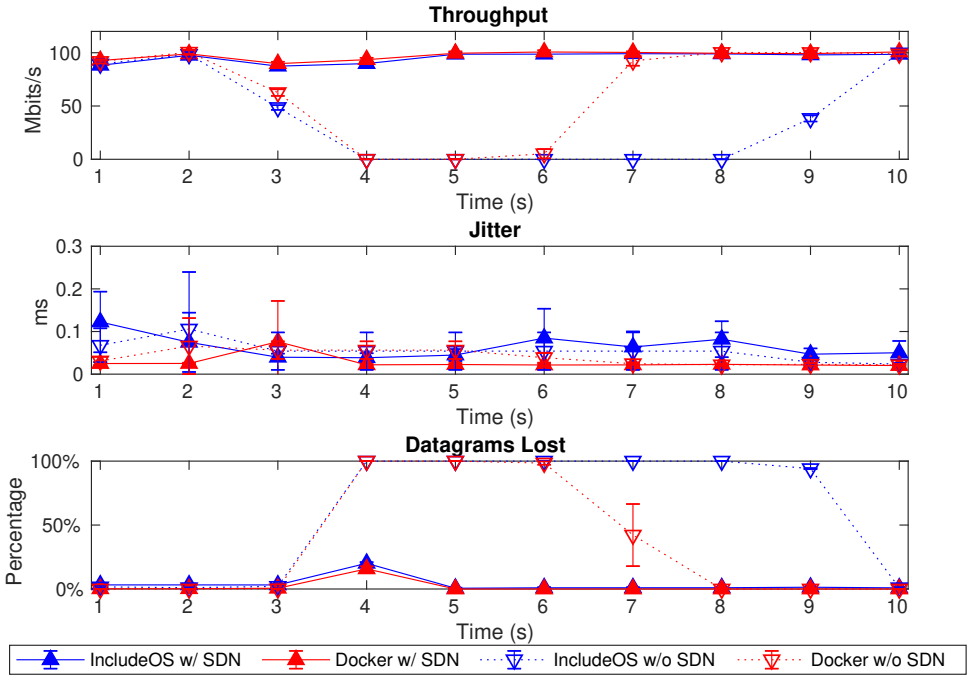

Figure 4.20: UDP (100 Mbps) Performance w/SDN Recovery Mechanism

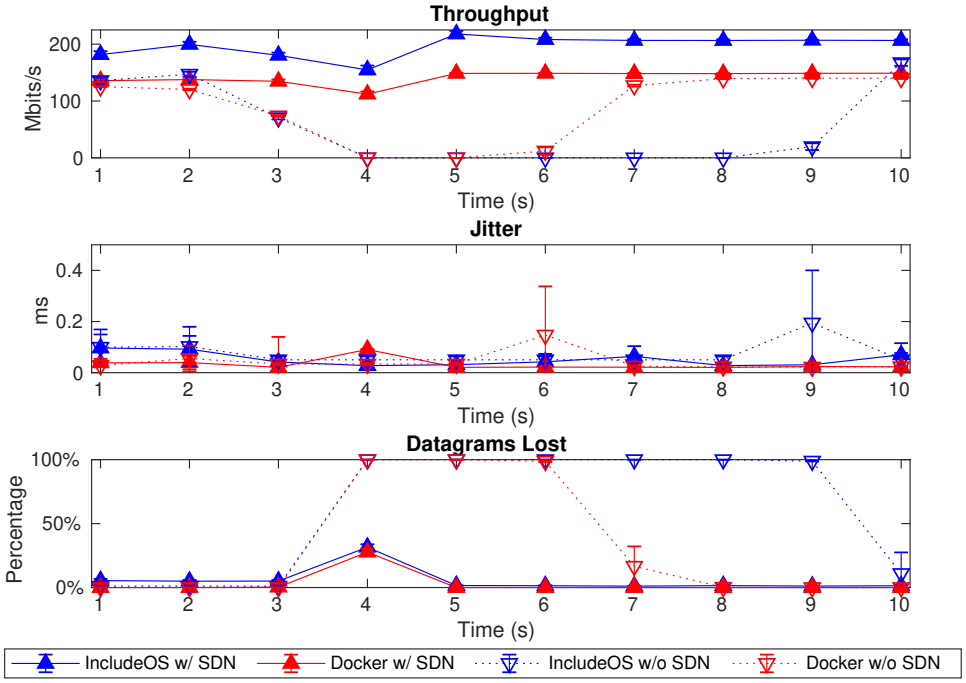

Figure 4.21: UDP (1 Gbps) Performance w/SDN Recovery Mechanism

4.7.2 Auto-Instantiation of Backup VNF

To test the performance difference when the backup VNF is automatically instantiated, an iPerf of UDP packets with 1 Gbps target bandwidth was performed during 25 seconds. The main VNF is instantiated as a IncludeOS unikernel and the backup VNF will be instantiated as a Docker container, since it instantiates roughly 2 seconds faster than the unikernel (Figure 4.1). The results are presented in Figure 4.22.

At Time = 10 sec., the main VNF reaches a critical point where 80% of the allocated memory is used, which triggers the instantiation of the backup VNF. The main VNF crashes at Time = 15 sec. and the OvS flow tables are updated at that time.

Even though that is a slight decrease in throughput in comparison to when the tests are performed with the two instances running in parallel, there is not a shattering difference. As for datagrams losses, both implementations lose roughly the same amount of datagrams.

Due to the nature of the data-center environment, which in this case resources are restricted, one could argue that the difference in throughput is worth it, as the backup VNF is only instantiated when it is strictly necessary.

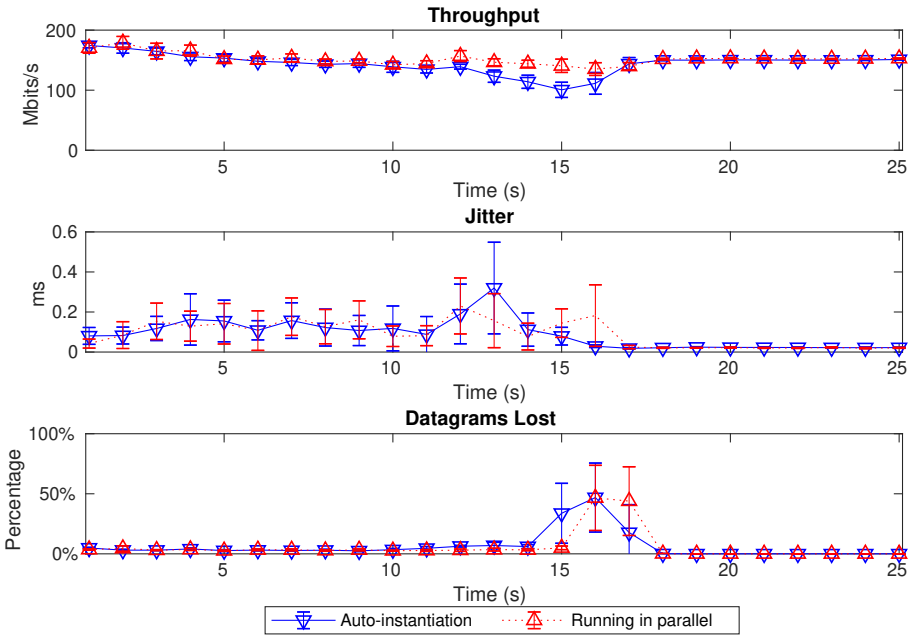

Figure 4.22: UDP (1 Gbps) Performance w/auto-instantiation of backup VNF

4.8 Summary

This chapter presented the tests and results of the VNFs performance for metrics such as instantiation times, latency, throughput and others. Also, the firewalls were evaluated on the update time to new versions and the update effect on traversing traffic, showing that the IncludeOS updates the fastest when the new versions are already pre-compiled and Docker presents better reactions to live updates. Finally, the reliability mechanisms were tested showing that Docker has a faster recovery time in all tests compared to the unikernel and that the SDN Recovery Mechanism ensures near zero downtime when a failure occurs.

The next chapter presents the thesis' conclusions, main contributions and future work.

5 | Conclusions

5.1 Conclusion

This thesis presented a comparison of performance between two distinct types of virtualization platforms - unikernels and containers - when instantiated as a VNF. A failure detection mechanism and a recover mechanism were also developed to achieve higher reliability and decrease the down-time of the VNF when a failure occurs.

From the results and analysis done in the previous chapter 4, we can conclude that when the VNF is instantiated in a Docker container, lower end-to-end latency and lower instantiation times for both stateless and stateful configuration are achieved. In terms of throughput, the container performs slightly better than the unikernel when TCP traffic is traversing the firewall. When the tests were done with UDP, both platforms presented the same throughput when the target bandwidth was set to 100 Mbps and the IncludeOS unikernel presented a smaller advantage in comparison to the container when the target bandwidth was set to 1 Gbps. Regarding TCP retransmissions and UDP datagram losses, Docker firewall performed better than the IncludeOS firewall in all tests. For jitter, both implementations presented approximately the same values with UDP 100 Mbps target bandwidth, but the unikernel jitter values for 1 Gbps target bandwidth were almost 3 times higher than the values retrieved for the container. As for the VNF update time, the number of rules added does not affect the update time for the IncludeOS firewall. This was expected since the IncludeOS is a library OS and every version needs to be compiled. Even though the number of rules affected the container update time, this platform presented an advantage since the update can be performed without stopping the running VNF, maintaining the baseline performance throughout the update.

The purpose of the recover mechanism was to minimize the down-time of the VNF. This was achieved with the conjunction of both failure and switching mechanisms, where Docker was proved to be the fastest platform for re-instantiation. This mechanism also demonstrated to be traffic independent as the amount of traffic passing through the firewall did not affect the time of re-instantiation.

To achieve zero downtime between failures, a SDN Recovery Mechanism was proposed. This mechanism proved to be essential for the reliability of the service as the results from the previous chapter 4 showed that with this mechanism, throughput values never drop to zero and fewer TCP retransmissions take place as well as fewer UDP datagrams are lost, when a failure is induced in the VNF.

In conclusion, Docker containers proved to be a better platform than IncludeOS unikernels for the instantiation of critical VNFs for the use-case presented in this work.

5.2 Contributions

This thesis execution resulted in a physical testbed that can be implemented as a baseline architecture for future critical M2M use-case scenarios for the ongoing "Mobilizador 5G - Components and services for 5G network" project. Still regarding the "Mobilizador 5G" project, contributions were also made for the "Deliverable D3.1 - M2M critical communications scenarios and analysis of architectures" report.

The work made on this thesis also resulted in the paper submission entitled "A Performance Comparison of Containers and Unikernels for Reliable 5G Environments" to the Institute of Electrical and Electronics Engineers (IEEE) Design of Reliable Communication Networks 2019 Conference, with the authors João Barraca Filipe, Flávio Meneses, A. U. Rehman, Daniel Corujo and Rui L. Aguiar.

5.3 Future Work

As future work, other more refined solutions including monitoring and operation assessment mechanism more tailored towards commercial environments could be implemented to further improve this type of services.

Also, other types of VNFs, such as DNSs, CDNs, IDSs could be compared between these two different virtualization technologies. Not only other VNFs could be tested, but other containers and unikernels could also be evaluated.

Another complement to the proposed architecture would be to take into account the scalability of the VNFs running in the data-center. Since applications and services can have fluctuating resources requirements, depending on external factors such as time of day, load demand and network conditions, it is crucial to prepare these critical and reliable VNFs with automatic scalability features according to performance requirements.

References

[1] NGMN Alliance, *5g white paper*, 2015.

[2] 5G PPP Architecture Working Group, «View on 5g architecture», 2016.

[3] P. Popovski, «Ultra-reliable communication in 5g wireless systems», in *1st International Conference on 5G for Ubiquitous Connectivity*, Nov. 2014, pp. 146–151. DOI: 10.4108/icst.5gu.2014.258154.

[4] Nokia Networks, «5g for mission critical communication–achieve ultra-reliability and virtual zero latency, white paper», 2018. [Online]. Available: http://www.hit.bme.hu/~jakab/edu/litr/5G/Nokia_5G_for_Mission_Critical_Communication_White_Paper.pdf.

[5] A. E. Kalør, R. Guillaume, J. J. Nielsen, A. Mueller, and P. Popovski, «Network slicing for ultra-reliable low latency communication in industry 4.0 scenarios», *ArXiv preprint arXiv:1708.09132*, 2017.

[6] 3GPP, «Feasibility study on new services and markets technology enablers for critical communications; stage 1», 3rd Generation Partnership Project (3GPP), TR 22.862, 2016. [Online]. Available: http://www.3gpp.org/ftp/Specs/archive/22_series/22.862/22862-e10.zip.

[7] H. Ji, S. Park, J. Yeo, Y. Kim, J. Lee, and B. Shim, «Ultra-reliable and low-latency communications in 5g downlink: Physical layer aspects», *IEEE Wireless Communications*, vol. 25, no. 3, pp. 124–130, Jun. 2018, ISSN: 1536-1284. DOI: 10.1109/MWC.2018.1700294.

[8] A. Jain, S. N. S, S. K. Lohani, and M. Vutukuru, «A comparison of sdn and nfv for re-designing the lte packet core», in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov. 2016, pp. 74–80. DOI: 10.1109/NFV-SDN.2016.7919479.

[9] E. Haleplidis, K. Pentikousis, S. Denazis, J. Salim, D. Meyer, and O. Koufopavlou, «Software-defined networking (sdn): Layers and architecture terminology (no. rfc 7426)», 2015.

[10] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, «Forwarding and control element separation (forces) protocol specification», Tech. Rep., 2010.

[11] M. Bjorklund, «Yang-a data modeling language for the network configuration protocol (netconf)», Tech. Rep., 2010.

[12] Open Networking Fundation, «Software-defined networking: The new norm for networks», *ONF White Paper*, vol. 2, pp. 2–6, 2012.

[13] OpenFlow Switch Specification, «Version 1.0. 0 (wire protocol 0x01)», *Open Networking Foundation*, 2009.

[14] OpenFlow Switch Specification , «Version 1.3. 5 (protocol version 0x04)», *Open Networking Foundation*, 2017.

[15] Red Hat Enterprise Linux, Inc., *What is virtualization?*, Red Hat Enterprise Linux, Inc. [Online]. Available: `https://www.redhat.com/en/topics/virtualization/what-is-virtualization` (visited on 10/16/2018).

[16] G. J. Popek and R. P. Goldberg, «Formal requirements for virtualizable third generation architectures», *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974, ISSN: 0001-0782. DOI: `10.1145/361011.361073`. [Online]. Available: `http://doi.acm.org/10.1145/361011.361073`.

[17] OpenSource, *What is virtualization?*, Red Hat Enterprise Linux, Inc. [Online]. Available: `https://opensource.com/resources/virtualization` (visited on 10/16/2018).

[18] KVM, *Main page — kvm,* [Online; accessed 24-October-2018], 2016. [Online]. Available: `https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792`.

[19] Red Hat Enterprise Linux, Inc., *What is kvm?*, Red Hat Enterprise Linux, Inc. [Online]. Available: `https://www.redhat.com/en/topics/virtualization/what-is-KVM` (visited on 10/16/2018).

[20] Unikernel Project, *Unikernels - rethinking cloud infrastructure.* [Online]. Available: `http://unikernel.org/` (visited on 10/16/2018).

[21] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, «Unikernels: Library operating systems for the cloud», in *Acm Sigplan Notices*, ACM, vol. 48, 2013, pp. 461–472.

[22] SysML, *Clickos -fast and lightweight network function virtualization.* [Online]. Available: `http://cnp.neclab.eu/projects/clickos/` (visited on 10/16/2018).

[23] LSUB, *Clive - removing (most of) the software stack from the cloud.* [Online]. Available: `http://lsub.org/ls/clive.html` (visited on 10/16/2018).

[24] Microsoft, *Drawbridge.* [Online]. Available: `http://research.microsoft.com/en-us/projects/drawbridge/` (visited on 10/16/2018).

[25] Galois, *Haskell lightweight virtual machine.* [Online]. Available: `https://galois.com/project/halvm/` (visited on 10/16/2018).

[26] IncludeOS, *Includeos.* [Online]. Available: `http://www.includeos.org/` (visited on 10/16/2018).

[27] Erlang, *Erlang on xen - at the heart of super-eastic clouds.* [Online]. Available: `http://erlangonxen.org/` (visited on 10/16/2018).

[28] Xen and Linux Foundation, *Mirage os - a programming framework for building type-safe, modular systems.* [Online]. Available: `https://mirage.io/` (visited on 10/16/2018).

[29] CLOUDIUS SYSTEMS, *Osv. designed for the cloud.* [Online]. Available: `http://osv.io/` (visited on 10/16/2018).

[30] Rumprun, *Rump kernels - you can make an omelette without breaking the kitchen.* [Online]. Available: http://rumpkernel.org/ (visited on 10/16/2018).

[31] runtime js, *Runtime.js - javascript library operating system for the cloud.* [Online]. Available: http://runtimejs.org/ (visited on 10/16/2018).

[32] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, «Includeos: A minimal, resource efficient unikernel for cloud services», in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov. 2015, pp. 250–257. DOI: 10.1109/CloudCom.2015.89.

[33] Google Cloud, *Containers at google - a better way to develop and deploy applications.* [Online]. Available: https://cloud.google.com/containers/ (visited on 10/16/2018).

[34] AWS, *What are containers?* [Online]. Available: https://aws.amazon.com/what-are-containers/ (visited on 10/16/2018).

[35] Linux Containers, *Lxc - introduction.* [Online]. Available: https://linuxcontainers.org/lxc/introduction/ (visited on 10/16/2018).

[36] R. Dua, A. R. Raja, and D. Kakadia, «Virtualization vs containerization to support paas», in *2014 IEEE International Conference on Cloud Engineering*, Mar. 2014, pp. 610–614. DOI: 10.1109/IC2E.2014.41.

[37] Cloud Foundry, *Cloud foundry concepts.* [Online]. Available: https://docs.cloudfoundry.org/concepts/ (visited on 10/16/2018).

[38] Virtuozzo, *Openvz - open source container-based virtualization for linux.* [Online]. Available: https://openvz.org/ (visited on 10/16/2018).

[39] M. Plauth, L. Feinbube, and A. Polze, «A performance evaluation of lightweight approaches to virtualization», *Cloud Computing*, vol. 2017, p. 14, 2017.

[40] Docker, *What is a container - a standardized unit of software.* [Online]. Available: https://blog.docker.com/2016/09/4-biggest-questions-docker-vmworld-2016/ (visited on 10/16/2018).

[41] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, «Network function virtualization: State-of-the-art and research challenges», *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, Firstquarter 2016, ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2477041.

[42] European Telecommunications Standards Institute, *Etsi gs nfv 002 v1. 2.1*, 2014.

[43] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, «Network function virtualization: State-of-the-art and research challenges», *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, Firstquarter 2016, ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2477041.

[44] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, «A view of cloud computing», *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010, ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672.

[45] P. Mell, T. Grance, *et al.*, «The nist definition of cloud computing», 2011.

[46] P. Gill, N. Jain, and N. Nagappan, «Understanding network failures in data centers: Measurement, analysis, and implications», in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 41, 2011, pp. 350–361.

[47] A.-A. Walla, «Live updating includeos while preserving state», Master's thesis, Faculty of Mathematics and Natural Sciences - UNIVERSITY OF OSLO, 2017.

Appendix-A: Sink Node

Dockerfile

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y \
  libc6 \
  libpcap0.8 \
  apparmor \
  libssl1.0.0 \
  libssl-dev \
  net-tools \
  traceroute \
  iptables \
  arping \
  ipcalc \
  iperf3 \
  netcat \
  inetutils-ping \
  curl dnsutils wget  vim ethtool tcpdump \
&& apt clean
COPY routes /routes
CMD /routes
```

ROUTES

```
#!/bin/bash
ip route del default via 192.168.0.1
ip route add 10.0.0.0/24 via 192.168.0.3
echo ""
echo "----Sink Node Ready----"
echo ""
ip route
bash
```

Appendix-B: Database

Dockerfile

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y \
  libc6 \
  libpcap0.8 \
  apparmor \
  libssl1.0.0 \
  libssl-dev \
  net-tools \
  traceroute \
  iptables \
  arping \
  ipcalc \
  iperf3 \
  netcat \
  inetutils-ping \
  curl dnsutils wget  vim ethtool tcpdump \
&& apt clean
COPY versions_docker /versions_docker #versions for the Docker Firewall
COPY versions_ios /versions_ios #versions for the IncludeOS Firewall
COPY http_server /http_server
CMD /http_server
```

http__ server

```
#!/bin/bash
python3 -m http.server
```

Appendix-C: Docker Firewall

Dockerfile

```
FROM alpine
RUN \
  apk --update add iptables coreutils bash
COPY start /start
CMD /start
```

start (Stateless - version 0)

```
#!/bin/bash
#Make the firewall stateless:
iptables -t raw -I PREROUTING -j NOTRACK
iptables -t raw -I OUTPUT -j NOTRACK
iptables -P INPUT DROP
iptables -P FORWARD DROP
iptables -P OUTPUT DROP
#Only allows tcp connections to whitelist IPs and 80/443 ports:
for i in {1..3}   #White list - 192.168.0.1-3
do
        iptables -A FORWARD -p tcp -d 192.168.0.$i --dport 80 -j ACCEPT
        iptables -A FORWARD -p tcp -d 192.168.0.$i --dport 443 -j ACCEPT
        iptables -A FORWARD -s 192.168.0.$i -j ACCEPT
        iptables -A FORWARD -s 192.168.0.$i -j ACCEPT
done
#allow udp
iptables -A FORWARD -p udp -j ACCEPT
#allow pings to pass through
iptables -A FORWARD -p icmp --icmp-type echo-request -j ACCEPT
iptables -A FORWARD -p icmp --icmp-type echo-reply -j ACCEPT
#allow pings from outside:
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
```

```
#allow pings from inside:
iptables -A OUTPUT -p icmp --icmp-type echo-request -j ACCEPT
iptables -A INPUT -p icmp --icmp-type echo-reply -j ACCEPT
echo "Firewall v0 Ready"
```

START (STATEFUL)

```
echo "Statefull Firewall"
rm current_version
wget 10.0.0.99:8000/versions_docker/current_version
chmod 777 current_version
./current_version
```

DOCKER-COMPOSE.YML

```
version: "3"
services:
  app:
    image: fw_stateful_img
    deploy:
      resources:
        limits:
          memory: 256M
    cap_add:
      - ALL
    networks:
      - outside
      - br
networks:
  br:
    external:
      name: br
  outside:
    external:
      name: outside
```

Appendix-D: IncludeOS Firewall

NaCl - configuration file

```
Iface outside {
        address:        10.0.0.2,
        netmask:        255.255.255.0,
        gateway:        10.0.0.1,
        index:          0
}
Iface inside {
        address:        192.168.0.3,
        netmask:        255.255.255.0,
        index:          1
}
Gateway myGateway {
        forward: firewallchain,
        outside_route: {
                net: 10.0.0.0,
                netmask: 255.255.255.0,
            Iface: outside
        },
        inside_route: {
                net: 192.168.0.0,
                netmask: 255.255.255.0,
                iface: inside
        },
        default_route: {
                net: 0.0.0.0,
                netmask: 0.0.0.0,
                nexthop: 10.0.0.1,
                iface: inside
        }
}
allowed_services: [52,80,443]
```

```
allowed_hosts: [10.0.0.2, 192.168.0.1-192.168.0.3]
Filter::IP firewallchain {
    Filter::ICMP {
        if (icmp.type ==  echo-request or icmp.type == echo-reply) {
           accept
        }
    }
    Filter::UDP {
           accept
    }
    Filter::TCP {
        if ((ip.daddr in allowed_hosts and tcp.dport in allowed_services) or
↪   ip.saddr in allowed_hosts) {
           accept
        }
    }
    log("Dropping from - ", ip.saddr, " -> ", ip.daddr," - Verify IP and/or
↪   Port\n")
    drop
}
```